NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

WEB SYNDICATION IN A MULTILEVEL SECURITY ENVIRONMENT

by

Avner Biblarz

March 2012

Thesis Co-Advisors:	Mark Gondree
	Zachary Peterson

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704–0188

1. REPORT DATE *(DD–MM–YYYY)* 29–3–2012	**2. REPORT TYPE** Master's Thesis	**3. DATES COVERED** *(From — To)* 2011-06-23—2012-03-29

4. TITLE AND SUBTITLE Web Syndication in a Multilevel Security Environment	**5a. CONTRACT NUMBER**
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) Avner Biblarz	**5d. PROJECT NUMBER**
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)	**10. SPONSOR/MONITOR'S ACRONYM(S)**
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited

13. SUPPLEMENTARY NOTES

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

14. ABSTRACT

In this thesis, we demonstrate the feasibility of a novel multilevel web application that merges the ability to share sensitive information with cutting-edge Web 2.0 communication paradigms: we develop a multilevel web aggregation service, allowing web content at various classifications to be gathered together and browsed. The architecture supports read-down across subscriptions, supports receiving near-real-time delivery of new *low* web content to *high* subjects and demonstrates several thoughtful, ergonomic user interfaces relevant in a multilevel security context. The architecture was prototyped and evaluated using the current Monterey Security Architecture (MYSEA) research system.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			**19b. TELEPHONE NUMBER** *(include area code)*
Unclassified	Unclassified	Unclassified	UU	95	

NSN 7540-01-280-5500

Standard Form 298 (Rev. 8–98)
Prescribed by ANSI Std. Z39.18

THIS PAGE INTENTIONALLY LEFT BLANK

WEB SYNDICATION IN A MULTILEVEL SECURITY ENVIRONMENT

Avner Biblarz
Civilian, Naval Postgraduate School
B.A., Economics, University of California at Berkeley, 2001
B.A., Applied Mathematics, University of California at Berkeley, 2001

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
March 2012

Author:	Avner Biblarz
Approved by:	Mark Gondree Thesis Co-Advisor
	Zachary Peterson Thesis Co-Advisor
	Peter J. Denning Chair, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

In this thesis, we demonstrate the feasibility of a novel multilevel web application that merges the ability to share sensitive information with cutting-edge Web 2.0 communication paradigms: we develop a multilevel web aggregation service, allowing web content at various classifications to be gathered together and browsed. The architecture supports read-down across subscriptions, supports receiving near-real-time delivery of new *low* web content to *high* subjects and demonstrates several thoughtful, ergonomic user interfaces relevant in a multilevel security context. The architecture was prototyped and evaluated using the current Monterey Security Architecture (MYSEA) research system.

THIS PAGE INTENTIONALLY LEFT BLANK

Table of Contents

List of Figures

THIS PAGE INTENTIONALLY LEFT BLANK

List of Tables

THIS PAGE INTENTIONALLY LEFT BLANK

List of Acronyms and Abbreviations

APC	Alternative PHP Cache
BLP	Bell and LaPadula
CDS	Cross-Domain Solution
CSS	Cascading Style Sheets
CMS	Content Management System
DAC	Discretionary Access Control
DoD	Department of Defense
DOM	Document Object Model
DTD	Document Type Definition
GD	Graphics Draw
GIF	Graphics Interchange Format
GIG	Global Information Grid
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IKE	Internet Key Exchange
IMAP	Internet Message Access Protocol
JPEG/JPG	Joint Photographic Experts Group
LAMP	Linux, Apache, MySQL and PHP
LAN	Local Area Network
MAC	Mandatory Access Control
MILS	Multiple Independent Levels of Security
MLS	Multilevel Security
MYSEA	Monterey Security Architecture
NIPRNet	Non-Classified Internet Protocol Router Network
NPS	Naval Postgraduate School
NRL	Naval Research Laboratory
OS	Operating System
PHP	PHP Hypertext Processor
PNG	Portable Network Graphics
PRISM	Program Replication and Integration for Seamless MILS
RDBMS	Relational Database Management System
RSS	Really Simple Syndication or RDF Site Summary

SIPRNet	Secret Internet Protocol Router Network
STOP	Secure Trusted Operating System
TCB	Trusted Computing Base
TPE	Trusted Path Execution
TSE	Trusted Services Engine
UI	User Interface
URL	Uniform Resource Locator
USG	United States Government
WebDAV	Web Distributed Authoring and Versioning
WWW	World Wide Web
XML	Extensible Markup Language
XMPP	Extensible Messaging and Presence Protocol
XSLT	Extensible Stylesheet Language
YUI	Yahoo! User Interface Library

Acknowledgements

This material is based on work supported by the National Science Foundation under Grant DUE–SFS Scholarships.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 1: Introduction and Background

Recently, current National Intelligence Director James Clapper, during his keynote address at the Center for Strategic & International Studies in Washington, D.C., remarked on the simultaneous urgency and care that must be considered in the context of information sharing:

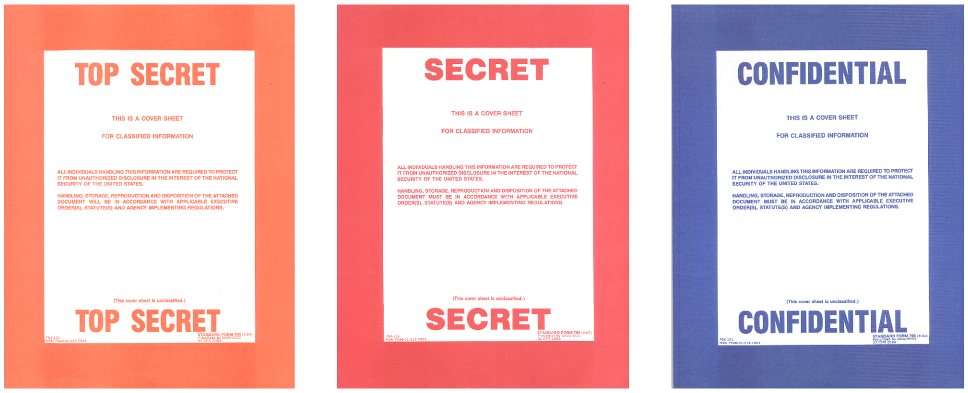
> "Information sharing has been a huge mandate for us all since 9/11. The notion of sharing is an interesting concept; it can be phenomenal, it can be dangerous. It depends on what is shared and with whom it's shared...Sharing must be done responsibly, seamlessly, and securely [1]."

Clapper's statements resonate particularly well with the motivation and long-range vision of the Department of Defense's Global Information Grid [2]. Multilevel secure systems attempt to facilitate this kind of controlled, information sharing. In this thesis, we demonstrate the feasibility of a novel multilevel web application that merges the ability to share sensitive information with cutting-edge Web 2.0 communication paradigms: we develop a multilevel web aggregation service, allowing web content at various classifications to be gathered together and browsed. The architecture supports read-down across subscriptions, supports receiving near-real-time delivery of new *low* web content to *high* subjects and demonstrates several thoughtful, ergonomic user interfaces relevant in a multilevel security context. The architecture was prototyped and evaluated using the current Monterey Security Architecture (MYSEA) research system.

We begin by discussing the development of multilevel security.

1.1 Mandatory Access Control and Multilevel Security

With the Second World War began the widespread practice of marking the sensitivity of pieces of information with classification labels [3, p.140]. Labels were based on the severity of the consequences that may result if that data fell into the wrong hands. This general data labeling strategy is still in wide use today within the DoD (see Figure 1.1).


TOP SECRET

THIS IS A COVER SHEET

FOR CLASSIFIED INFORMATION

ALL INDIVIDUALS HANDLING THIS INFORMATION ARE REQUIRED TO PROTECT IT FROM UNAUTHORIZED DISCLOSURE IN THE INTEREST OF THE NATIONAL SECURITY OF THE UNITED STATES.

HANDLING, STORAGE, REPRODUCTION AND DISPOSITION OF THE ATTACHED DOCUMENT WILL BE IN ACCORDANCE WITH APPLICABLE EXECUTIVE ORDER(S), STATUTE(S) AND AGENCY IMPLEMENTING REGULATIONS.

(This cover sheet is unclassified.)

TOP SECRET

703-101
NSN 7540-01-213-7901

STANDARD FORM 703 (8-85)
Prescribed by GSA/ISOO
32 CFR 2003

SECRET

THIS IS A COVER SHEET

FOR CLASSIFIED INFORMATION

ALL INDIVIDUALS HANDLING THIS INFORMATION ARE REQUIRED TO PROTECT IT FROM UNAUTHORIZED DISCLOSURE IN THE INTEREST OF THE NATIONAL SECURITY OF THE UNITED STATES.

HANDLING, STORAGE, REPRODUCTION AND DISPOSITION OF THE ATTACHED DOCUMENT MUST BE IN ACCORDANCE WITH APPLICABLE EXECUTIVE ORDER(S), STATUTE(S) AND AGENCY IMPLEMENTING REGULATIONS.

(This cover sheet is unclassified.)

SECRET

704-101
NSN 7540-01-213-7902

STANDARD FORM 704 (8-85)
Prescribed by GSA/ISOO
32 CFR 2003

CONFIDENTIAL

THIS IS A COVER SHEET

FOR CLASSIFIED INFORMATION

ALL INDIVIDUALS HANDLING THIS INFORMATION ARE REQUIRED TO PROTECT IT FROM UNAUTHORIZED DISCLOSURE IN THE INTEREST OF THE NATIONAL SECURITY OF THE UNITED STATES.

HANDLING, STORAGE, REPRODUCTION AND DISPOSITION OF THE ATTACHED DOCUMENT MUST BE IN ACCORDANCE WITH APPLICABLE EXECUTIVE ORDER(S), STATUTE(S) AND AGENCY IMPLEMENTING REGULATIONS.

(This cover sheet is unclassified.)

CONFIDENTIAL

705-101
NSN 7540-01-213-7903

STANDARD FORM 705 (8-85)
Prescribed by GSA/ISOO
32 CFR 2003

Figure 1.1: Sample forms used to label in-transit, marked physical data within the U.S. Department of Defense.

1.1.1 Mandatory Access Control and Information flow policy

A system protecting labeled data gives a *subject* access to an *object* based on an access control *policy*. Unlike access control systems where policies are controlled by the user (i.e., discretionary access control [DAC] systems), in a system with data labeled according to sensitivity, access must be enforced according to a global policy on information flow. For example, a subject with *Secret* clearance may access data labeled *Secret* and *Confidential*, but not data labeled *Top Secret*. These labels are maintained by the system: changing the label or copying *Top Secret* data into a *Secret* container must be prevented by the system at all times. This concept forms the basis of a *mandatory access control* (MAC) policy.

The set of sensitivity labels to which mandatory information flow policies apply form the points of a lattice [4]. The structure of the lattice defines a dominance relationship between labels: $\mathscr{L}_i$ dominates $\mathscr{L}_j$ when a subject with $\mathscr{L}_i$-clearance can read data with label $\mathscr{L}_j$; we denote this dominance relationship by $\mathscr{L}_j \leq \mathscr{L}_i$. There may be points on the lattice that are incomparable (i.e., it is not the case that $\mathscr{A} \leq \mathscr{B}$ or $\mathscr{B} \leq \mathscr{A}$), which corresponds to the typical use of sensitive, compartmentalized classifications. Bell and LaPadula formalized a model describing data confidentiality and information flow in this context, called the Bell-LaPadula (BLP) policy model [5]. The BLP policy is a MAC policy comprised of two rules: (i) the simple security property, where a subject may not read information at a level higher than her clearance, and (ii) the confinement property, where a subject may not write to levels dominated by her clearance.

1.1.2 Multilevel Secure Systems

A multilevel secure (MLS) system can manage arbitrarily labeled information in accordance with a mandatory security policy, such as the BLP policy. The system must perform access checks and only grant access to authorized users. MLS systems are designed to withstand exploit by untrustworthy computer programs, i.e., Trojan Horses, which may attempt to copy sensitive information from higher to lower levels. There are several tradeoffs to consider when designing a *system architecture* capable of supporting an MLS policy [6]. We briefly discuss some of these issues.

Replication-based architectures utilize standalone, single-level components strictly separated by security level. The Multiple Independent Levels of Security (MILS) architecture is an example of such a design, which implements separation by level using a separation kernel [7]. This strict isolation may make a MILS system easier to formally evaluate compared to other systems, but may also result in expenses and inefficiencies, e.g., related to duplication and maintenance. Strict separation by level certainly prevents information flows that may disobey MAC policy, but also prevents others that are *allowed* under a BLP policy.

In replication architectures, transferring data from one single-level domain to another typically requires a policy-enforcing (i.e., trusted) mechanism, called a Cross-Domain Solution (CDS). Examples of CDS products include guards and data diodes. The Naval Research Laboratory (NRL) Network Pump [8] allows messages from a low sensitivity level to be sent to a higher sensitivity level, and prohibits information flow in the reverse direction. The difficulty in building and evaluating these types of systems is in limiting the data exchanged to only permitted flows [9].

Our focus in this work is on MLS systems capable of supporting an arbitrary MAC policy, and not simply a separation policy. Hinke proposes a system architecture where a high-assurance server is the locus of policy enforcement, through which single-level clients access resources in accordance with policy [10]. We are particularly interested in systems that are efficient in the presence of huge policy lattices, i.e., the so-called *gazillions* problem [11]. Next, we explore applying MLS concepts to web services.

1.2 MLS Web Applications

Since its development, the Internet and the World Wide Web (WWW) have been in a near-constant state of evolution. The Web has evolved into a vast information-sharing system that is less about static, linked Hypertext Markup Language (HTML) content and more about services for dynamic and collaborative sharing. The most recent set of dynamic web technologies and new modalities of user interaction with the WWW are collectively referred to as *Web 2.0*. Web 2.0, in part, includes new ways users interact with information, publish information and consume information. This includes recent web publishing technologies, such as wikis, blogs, social networks, and e-commerce.

It is natural to consider the role of web services in a multilevel environment, before considering web syndication in that context. Indeed, several architectures have been proposed for delivering content of different classifications to remote end-users using HTTP-based protocols. This includes replication-based architectures (i.e., offering single-level web services at a few levels) and architectures offering web services on-demand, at any level.

In replication-based architectures, MLS web services are implemented by an HTTP-based front-end to a trusted, content distribution service, whose role is to combine and replicate data from each domain. For example, Galois' Trusted Services Engine (TSE) implements a high-assurance Web Distributed Authoring and Versioning (WebDAV) server that supports automated content merging and replication across domains. The system supports a multi-level wiki (ML Wiki), backed by the TSE, which provides merged views of wiki pages to users, according to a BLP policy. Relatedly, the Program Replication and Integration for Seamless MILS (PRISM) engine also implements a CDS service for filtering and merging single-level content into a multilevel view [12]. PRISM supports a multilevel wiki, leveraging a special remote file system (*mlsfs*) that, again, replicates and merges data across levels.

Generically, cross-domain guards may be used to connect single-level networks, delivering (either explicitly or implicitly) labeled content from one domain to another [13–15]. Commercial Extensible Markup Language (XML) guards have been developed to automatically declassify or filter messages from high networks to low networks, such as Lockheed Martin's Radiant Mercury guard and BAE's DataSyncGuard. Guards like these act as the locus of enforcement in some architectures, providing web services between networks of different classifications.

Web services running as unprivileged subjects in a multilevel environment—one in which they are allowed access to some data and not others, based on MAC policy—can be designed (or re-factored) to be *multi-level aware* (MLS-aware), to function appropriately when restricted to read-only access to some of their resources. In MLS-aware software, system resources are labeled and their access mediated by a Trusted Computing Base (TCB). The web server software

Figure 1.2: Icon associated with RSS syndicated content.

itself performs no MAC enforcement, but is constrained by policy. When the TCB is an operating system, the granularity of labeling is typically at the file level. MLS-aware web servers [16] and wiki services [17] have been developed in this general framework. Functionally, we view MLS-aware web services as a least-privilege simulation of a trusted MLS web service; architecturally, however, the distinction between MLS-aware web services and MLS web services is significant (e.g., smaller TCB, no possibility of downgrading/leaking, no responsibility for filtering). We will often refer to MLS-aware services as MLS services, eliding these (important) architectural details. Next, we hone in on the specific web technology upon which the rest of our work focuses: the syndication of web content.

1.3 Web Syndication

Web syndication technologies, such as RSS and Atom, are a powerful method of distributing and consuming new web content. RSS most commonly stands for "really simple syndication" and is a standard that defines a structured document that describes metadata for published content. RSS is currently in version 2.0, though previous versions are still widespread. An RSS feed is an XML document, typically with file extension .xml or .rss.

For better or worse, there is a great deal of flexibility as to what an RSS file may contain. The only required content is the address of the website to which the RSS feed pertains, the feed title and description. These required elements are found within the `<channel>` tag of the XML file. Optional, though commonly used, the `<channel>` tag includes `<item>` tags. Each item must contain at least a title, description, and link (see Figure 1.3).

Images are a significant factor in the design of websites and may, themselves, provide valuable information. Surprisingly, RSS provides no dedicated "image" field to allow for embedded images. Various attempts have been made to remedy this problem: Yahoo! developed the Yahoo! Media RSS module [18], and RSS v2.0 introduced the "enclosure" field. Neither, however, allow the web publisher to embed anything other than thumbnail images in the actual feed. Common practice is to embed HTML into the RSS file directly. This process serializes HTML in a form that will be locally re-interpreted as HTML, rather than part of the RSS XML

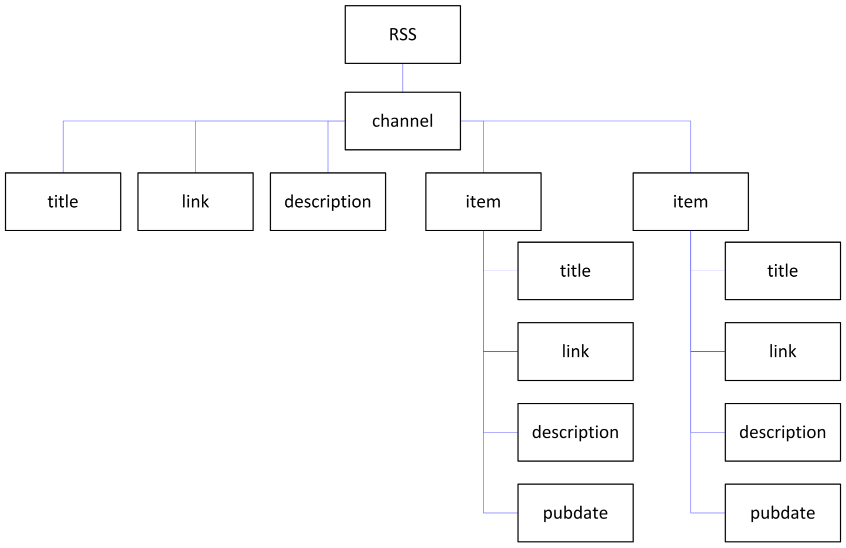


Figure 1.3: Sample RSS Schema with two items.

structure. Thus, `<img src="http://www.example.com"/>` becomes

```
&lt;img src=&quot;http://www.example.com&quot; /&gt;
```

There are many other elements that can be included in RSS feeds: audio (i.e., podcasts), geographic data, data "categories" and many others. A general lack of consistency, however, in the use of these fields greatly complicates the creation of software that aggregates and presents the data. If, for example, only a small fraction of feeds have a "category" tag, most feeds would end up labeled "Uncategorized." Thus, the utility of new data fields hinges greatly on community adoption and use, and lack of required elements weakens the capabilities of feed aggregators to consume and organize data.

In contrast to RSS, the Atom syndication format has more required fields and better use of timestamps. As a late entry, however, Atom missed out on critical branding opportunities. Today, both formats are used, though RSS dominates [19, p.131]. In general, syndication tends to be colloquially referred to as RSS, regardless of whether the Atom or RSS standard is utilized.

1.4 Feed Aggregation

Informally, a *feed aggregator* (or aggregator) is software that periodically polls for feeds describing new syndicated content from a variety of sources. Aggregation software has gained popularity, following the rise of blogging and new forms of web publishing. For frequently changing content (e.g., daily news websites), aggregators provide a single point of entry for a user to access and manage large volumes of web content. For infrequently published content (e.g., low-volume blogs), aggregators automate the repetitive tasks associated with polling for new content, allowing users to follow a large number of sources with little overhead. There are many types of software designed to consume syndicated content, reflecting the multitude of ways data is read, explored, and used on the Web. Next, we characterize some trends based on a survey of syndication-related projects (see Table 1.1 for a summary of those projects).

1.4.1 Personal Desktop Aggregators

We characterize a desktop aggregator as any software that resides on a client and aggregates feeds from various sources for a user. Some software extends or interoperates with popular e-mail programs, web browsers, and operating systems. Many popular e-mail clients and web browsers already incorporate the ability to fetch and present syndicated content. Other aggregators are standalone, dedicated applications, like BlogBridge [20], RSSOwl [21], AmphetaDesk [22], and AgileRSS [23]. These allow the user to navigate content using a graphical interface, keep state on behalf of the user (marking items as read or unread), and allow the user to organize, search, filter, and tag feed items. Some software presents feed items in the form of a news ticker, showing summaries of syndicated content as feed updates are detected.

1.4.2 Podcatchers and other Media Desktop Aggregators

Since RSS 2.0 and Atom 1.0, these syndication formats support descriptions for external media (e.g., MP3 audio, MP4 video, BitTorrent files). This is an enabling technology for a variety of new publishing trends, like podcasting, vlogging, and photoblogging. Due to the typical size of the syndicated media (on the order of megabytes), these aggregators are client-side applications designed to download new content slowly, in the background. Additionally, these applications typically integrate support for handling the target media. For example, podcatchers—software for managing feeds associated with syndicated audio content, or *podcasts*—typically include support for playing the audio. Broadcatching—a term used to describe the wide distribution of syndicated media, like Internet television—is sometimes achieved by advertising files using RSS and distributing the files using the peer-to-peer BitTorrent protocol; thus, some BitTorrent clients include support for retrieving RSS feeds and interpreting enclosures that contain torrent descriptions.

1.4.3 Server-Side Personal Aggregators

Given the proliferation of web-enabled devices, it's natural to consider migrating the personal aggregator concept from the desktop into the cloud. Most of these are server-resident applications accessed using a web browser, allowing a user to aggregate syndicated content, organize and tag feeds, and mark feed items as read or unread. Others—like rss2mail [24], Newspipe [25], and feed2imap [26]—are administered via a web interface, collect syndicated content, and then distribute it via e-mail. Server-side aggregators often utilize schedulable logic (e.g., cron jobs) to update and cache feeds, separating content retrieval from presentation. Aggregators that do not fetch and cache content in advance of presentation typically suffer performance penalties: when feeds are fetched in real-time, the least responsive feed source will necessarily act as a bottleneck.

1.4.4 Community Aggregators / Single Page Aggregators

Some server-side web applications aggregate content and present it to visitors, eschewing the complexities associated with managing per-user state (marking items read or unread) or managing a set of unique feeds for each user. Extensions or plugins to popular CMS software and online forum software, for example, add value to these applications by enabling a set of feeds to be displayed for all visitors. Some software is deployed as a standalone website which aggregates feeds, with the intention of adding value to those sources (like item ratings, comments, or customizable interfaces); PopUrls [27], a site which displays a collection of the web's most visited social news sites, is a prominent example. Other websites collect feeds targeted towards some special interest community; for example, Planet Mozilla [28] collects feeds associated with Mozilla community news and Mozilla developer blogs.

1.4.5 Re-Aggregators

Some server-side software not only aggregates feeds, but also re-publishes the aggregated items as a new feed. These have been described as RSS re-aggregators or RSS-mashups, because they take data from multiple sources and present a new, composed service. Examples of these services include FeedWeaver [29], xFruits [30], BlogSieve [31], FeedRinse [32], and Yahoo! Pipes [33]. Some services support advanced filtering, search, notification, and other user-definable process logic during feed generation.

In our next chapter, we explore necessary components of a multilevel web content aggregator.

Project	*Reference*	*Basic Characterization*	*Open-Source License*	*Dependencies*	*Multiple Users*	*Organize / tag feeds*	*OPML Support*	*Mark items as "read"*	*Generate new feeds*	*Schedulable caching*
Bloglines	[34]	Section 1.4.3	–		X	X	X	X		
Google Reader	[35]	Section 1.4.3	–		X	X	X	X	X	
FeedShow	[36]	Section 1.4.3	–		X	X	X	X		
feed on feeds	[37]	Section 1.4.3	GPL	PHP 4.3.2+,MySQL	–		X	X		/
Tiny Tiny RSS	[38]	Section 1.4.3	GPL	PHP 5+,MySQL/Postgres	–	X	X	X	X	–
zFeeder	[39]	Section 1.4.3	GPL	PHP 4.2+	–	X	X	–		/
lylina	[40]	Section 1.4.3	GPL	PHP 5.2+,MySQL	X	X	X	–		/
Rnews	[41]	Section 1.4.3	GPL	PHP 4.3+,MySQL	X	X	X	X		/
Urchin	[42]	Section 1.4.5	various	Perl, CPAN, MySQL	–		X	–	X	
Lilina	[43]	Section 1.4.4	GPL	PHP 5.2+	–		X	–	X	/
Moonmoon	[44]	Section 1.4.4	BSD	PHP 5+	–		X	–	X	–
My News Crawler	[45]	Section 1.4.4	–	PHP, MySQL	–	X		–	X	–
PHP RSS Reader	[46]	Section 1.4.4	–	PHP, MySQL	–	X		–	X	–
PlanetPlanet	[47]	Section 1.4.4	PSF	Python 2.2+, Berkeley DB	–	–		–	X	–
rawdog	[48]		GPL	Python 2.4+	–	X	X	–	–	X
curn	[49]		BSD	Java, JAF, Jakarta BSF	–			–	–	X
gPodder	[50]	Section 1.4.2	GPL		–	X	X	X	–	/
BlogBridge	[20]	Section 1.4.1	GPL		–	X	X	X	–	/
RSSOwl	[21]	Section 1.4.1	EPL		–	X	X	X	–	/

Table 1.1: A selection of aggregation software: a brief features survey. In the matrix above: (X) supported feature; (/) partial support, (-) no support or not applicable; (BSD) a BSD-like license; (EPL) Eclipse Public License; (GPL) GNU Public License; (PSF) Python Software Foundation License; blank entries are unknowns.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 2:
Requirements and Design

"A frog in a well cannot conceive of the ocean."

–Zhuangzi [51]

We introduce our method of MLS Aggregation in a multilevel environment. In particular, our aggregator is a server-resident news application (Section 1.4.3) capable of handling and delivering syndicated content from sources of different classifications. We call this the **MLS News Reader** application. While an aggregator with this functionality can be attained in a variety of multilevel architectures, we target the Monterey Security Architecture (MYSEA) research platform for prototype development and experimentation, as it meets many of our system requirements. Next, we describe our system requirements, our target architecture, and the concept of operations for an MLS News Reader.

2.1 High-Level Requirements

The following are requirements for our MLS News Reader application, following from popular secure design principles and from our target architecture:

1. *Minimize the introduction of privileged components.*
   In particular, a "trusted aggregator" is undesirable. Software should defer to the MAC and DAC enforcement provided by the TCB.
2. *Minimize the introduction of new security-relevant components.*
   Software should defer to the authentication and single sign-on mechanisms provided by the TCB.
3. *Software should not require any persistent client-state be maintained.*
   A server-resident web application should support stateless clients.
4. *Software cost should be minimal.*
   Cost of ownership should not be prohibitive.
5. *Interfaces should be intuitive and user friendly.*
   System behavior should be consistent with user expectations and principles of ergonomic security should be obeyed.

6. *Software should scale gracefully, to function with complex policies on the order of hundreds of security levels.*
   On the one hand, this may be viewed as an extreme requirement that serves merely to distinguish true MLS designs from weaker, multiple single-level designs. On the other hand, it motivates designs capable of addressing the *gazillions* problem, which is endemic to those multiuser MLS systems in which security compartments are commonplace.

2.2 Trusted Computing Base

We have selected the Monterey Security Architecture (MYSEA) design as our Trusted Computing Base (TCB). MYSEA allows users to take advantage of policy-constrained application services from thin-client workstations [52, 53]. *Policy-constrained*, or MLS-aware, means that an application has been modified to run in a multilevel environment without requiring extraordinary privileges, so it is both fully functional and constrained by the security policy. Thus, as stated in our requirements, we need not create a "trusted aggregator." This allows us to re-use and modify low cost, open source software to build untrusted applications that run at arbitrary security levels.

Our requirements, along with our choice of MYSEA, play a pivotal role in how we handle feeds. In particular, our first requirement limits the introduction of new trusted subjects, like XML guards, trusted XML query engines, and trusted web services. As we see below, this limits the granularity at which the system can label feed data.

2.2.1 Feed as a Collection of Labeled Objects

Highly granular access control policies for XML documents have been proposed [54–61], typically in the context of interpreting the XML document as a database and the XML query-execution engine as the locus of policy enforcement. Previous work has considered several variants of policy and labeling rules: labeling at the granularity of the XML element; labeling the XML schema or DTD; how to handle unlabeled items; policy inference for partially labeled documents (e.g., top-down label propagation, bottom-up label propagation, most-specific-takes-precedence propagation); how to resolve conflicting policies or inconsistently labeled documents; etc. The following questions must be strategically considered: Should a document containing unauthorized information (with respect to a user's clearance) be filtered, or should access to the entire logical document be denied? How can one filter or suppress those unauthorized elements that are required, per the schema? These are issues that affect DAC and MAC policies equally, and we do not summarize strategies that have been proposed to handle them

```
<?xml version="1.0" encoding="UTF-8" ?>
<rss version="2.0">
<channel>
    <title>Bobs International Law Blog</title>
    <description>International Surveillance</description>
    <link>http://mlsserver.org/~bob/law/blog/</link>
    ...
    <item>
        <title>Re: Recent Ruling in Hague Case</title>
        <summary>Hon. Muellers final ruling</summary>
        <link>http://mlsserver.org/~bob/law/blog/03/08/131502.html</link>
        ...
        <securitylabel><label>SECRET</label></securitylabel>
    </item>
    <item>
        <title>Excellent Banana Bread Recipe</title>
        <summary>The first secret is caramelized walnuts...</summary>
        <link>http://mlsserver.org/~bob/law/blog/03/03/203045.html</link>
        ...
        <securitylabel><label>UNCLASSIFIED</label></securitylabel>
    </item>
</channel>
</rss>
```

Figure 2.1: A simplified, notional feed with elements of different sensitivity labels.

here. In general, regardless of the specific XML labeling strategy being utilized, these schemes each allow the XML document to be traversed as a tree of nodes (i.e., DOM nodes) and, if the scheme is unambiguous, one can determine the security label of each node.

General proposals have been made to bind security labels to XML elements [62], as have proposals for extensions to specific XML standards. Specifically, Extensible Messaging and Protocol (XMPP) extensions supporting security labels [63] and Publish-Subscribe delivery [64] allow XMPP to be used as a transport for labeled RSS and Atom feeds. Figure 2.1 shows a notional example of an RSS feed with items that are associated with different security labels.

2.2.2 Feed as a Single Labeled Object

Instead of labeling objects within a feed document, we may treat each RSS or Atom feed document as a single labeled object. In this model, labels within the feed are unnecessary. Instead, the entire feed carries an effective label, matching that of the (untrusted) subject who wrote the content. This model allows untrusted subjects to generate feed content, and obviates the need for a trusted XML labeling service.

This model facilitates an architecture in which general-purpose trusted subjects, like a high-assurance operating system, manage the feeds. It then becomes possible to design an MLS RSS/Atom Aggregator following the Hinke-Schaefer (a.k.a kernelized) MLS RDBMS architectures [65], wherein (untrusted) application services get access to resources that are labeled and managed by a high-assurance kernel.

Given that we do not want a trusted XML labeling service, it follows directly that our prototype will not support XML structures with heterogeneous node labels (as in Section 2.2.1). Instead, data is labeled at the granularity supported by the underlying TCB, i.e., MYSEA. The MYSEA system itself inherits object labels from an underlying, high-assurance operating system, in this case BAE's Secure Trusted Operating System (STOP). Thus, RSS/Atom feeds will be labeled at the file granularity.

2.3 Concept of Operations

In order to validate a user's credentials, MYSEA uses a Trusted Path Extension (TPE) device. When a user logs in, the TPE passes the user's credentials to the MYSEA Server. The server validates the login attempt and instructs the TPE whether to allow or deny access to the network. In negotiating a session level, the TPE passes the user's request to the server for a decision. After a successful login and session level negotiation, the TPE allows the user to access the MLS LAN, the MYSEA Server and its services. To meet object reuse requirements [66], client state is purged at the end of each session, and data created or modified on the clients is stored on the MYSEA Server.

After a user establishes a session at a level, she is able to access the MLS News Reader web application using a web browser from her thin client. This is a server-side, personal news reader (aggregator), which presents a way to navigate the contents of RSS/Atom feeds for any level her session dominates, i.e., from any of the single-level sources and MLS-aware services. The user will be able to add or delete feeds associated with her current level. The service will be able to read-down and access subscriptions made at lower levels, during previous sessions. The user will be able to browse all of her subscriptions (by level, by feed, chronologically, etc.) and the data associated with those feeds. Thus, a user at *Secret* will be able to view items in both *Secret* and *Unclassified* feeds, e.g., using the same web interface to read syndicated *Unclassified* CNN news content and syndicated *Secret* blog content, simultaneously.

Despite the fact that data are labeled at the granularity of the feed, our read-down requirements will cause data from different levels to be intermixed. For example, chronological browsing may cause an item from a wiki's *Secret* Atom feed to appear between two items from a blog's *Unclassified* RSS feed. It follows that the MLS News Reader application must unambiguously mark the data it displays (with advisory labels) using the label of the source (feed), i.e., delineating where the *Secret* data begins and ends.

Next, we describe the design of our MLS News Reader application, based on these preliminary high-level and functional requirements.

2.4 Design Overview

The MLS News Reader application design leverages an existing MLS web service. On MYSEA, MLS web services are implemented by spawning, on demand, an unprivileged Apache process at the level of the requestor's user session, to service the end-user's requests. The MLS News Reader will read-down to access a set of single-level *state slices*. Each slice is a labeled object managed by MYSEA, holding state used by the application. In particular, the slice holds a list of the user's subscriptions. This state may be implemented in a variety of ways: it could be a structured file like XML, or a flat-file database such as those utilized by SQLite. It could hold arbitrary feed or reader state, including alternative feed titles, feed groupings and read/unread per-item state.

For each subscription read from a slice, the MLS News Reader retrieves feed items from a public, shared, *per-level feed cache* maintained by MYSEA. This cache holds all recent feed items and some embedded feed content (e.g., images, favicons). As previously remarked, feed caches are used in several aggregator designs, as they allow items to be efficiently retrieved without suffering performance penalties associated with unresponsive, or intermittently failed, remote content producers.

For any feed in a slice at the subject's session level—i.e., the slice that is both readable and writable—if the cached feed is *expired* then the cache will be updated. For example, a *Secret* subject has the ability to update the *Secret* cache, since all feeds in the *Secret* slice are interpreted as feeds that are either (i) from sources on the *Secret*-level backend network, i.e., the SIPRNet, or (ii) from *Secret* subjects in the MLS LAN. Subjects at *Secret* will only be able to subscribe to feeds of this kind, since they cannot write requests to subjects at lower levels.

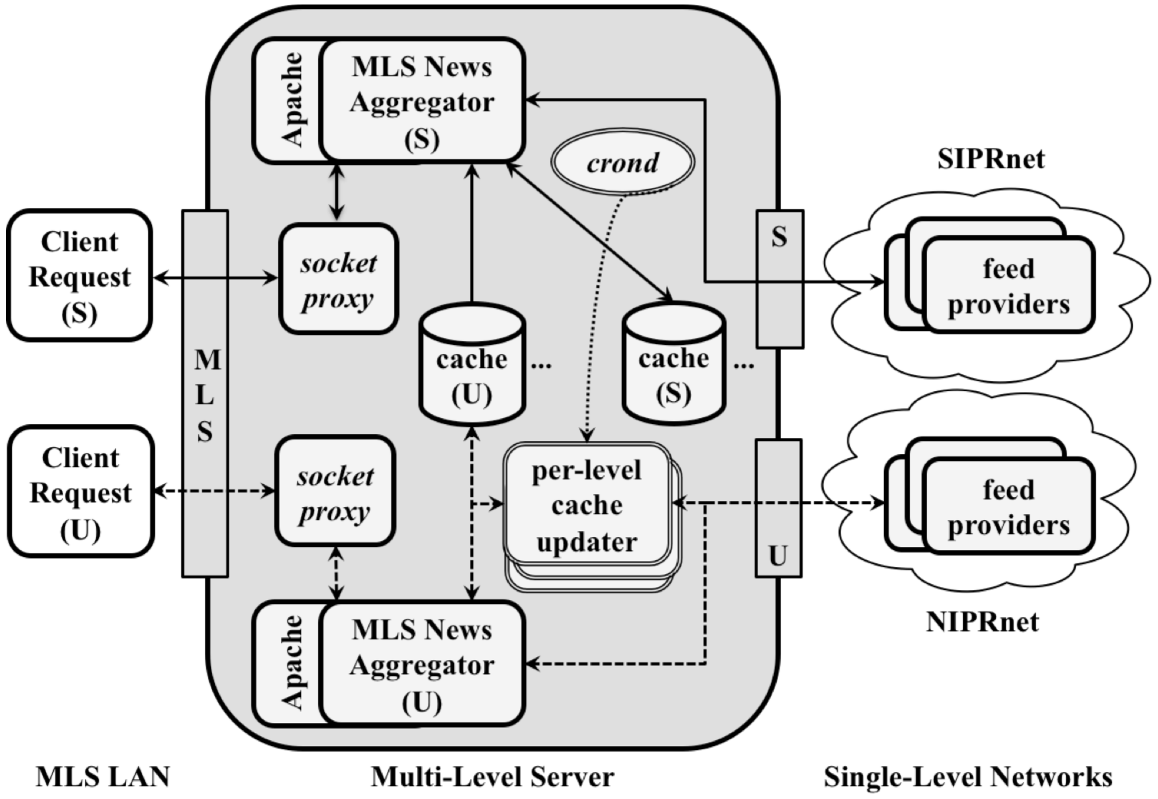


Figure 2.2: Information flow among components in the MLS News Reader design.

In addition, at each level, a scheduled, unprivileged job will periodically poll all slices at its level and update the cache with new feeds. For example, an *Unclassified cache updater* job will read all the *Unclassified* slices and attempt to update the item and image *Unclassified* cache based on these subscriptions. Likely, this unprivileged job may require some DAC exception, to read slices owned by arbitrary users; alternatively, slices can be written with appropriate permissions for this cache update daemon, e.g., readable by a *news-update* group.

Starting an unprivileged process in the absence of a MYSEA user session is, however, not currently supported. We envision a new *trusted cron service* that would be able to spawn untrusted processes at any level, at (deterministic, scheduled) intervals. Such a service would allow a *Secret* user to enjoy a channel-free mechanism to see "fresh" content in the *Unclassified* cache, while remaining logged in at *Secret*.

The MLS News Reader design is summarized in Figure 2.2, which highlights information flow between the main trusted (italicized) and untrusted components. Our prototype implementation of this design leverages MYSEA. We remark that our prototype implementation deviates from our architectural design in a few notable ways: (i) some of the design remains unimplemented (components with hollow borders are planned but not yet prototyped) and (ii) some information flows have changed. In particular, in our implementation, the socket proxies mediate access between the untrusted subject and *all* network interfaces, even those at the subject's level. We refer the reader to other sources for an account of MYSEA's design and implementation [52,53].

2.4.1 Interface Requirements

One of the most significant challenges of any feed aggregator is the *presentation* of feed data: how does one present the user with a potentially massive amount of information, on the order of thousands of feeds, each with a variable number of items and a diverse set of metadata? We describe some of the user interface design targets for our prototype MLS News Reader.

Advisory Labels

The MLS News Reader navigation must be able to handle an arbitrary number of security levels and feeds. Each feed must be clearly labeled with its security label. Artifacts from each feed, such as story items and images, must also be identified by an appropriate label. The navigation should be organized such that the user knows under which security level the feed is contained. In our design, the user should have multiple ways of viewing feed data.

Per-Level View

The MLS News Reader should provide the user with a variety of methods for sorting data. In particular, the user should be able to view the most recent items from all feeds at her current session level, or at any level her session dominates.

Chronological View

Like many other aggregators, our MLS News Reader should allow users to browse feed items chronologically. Items from a user's subscriptions (i.e., from feeds at different classifications) should be merged into a unified, multilevel view across time.

Snapshot View

While a chronological view may be natural, it can hide potentially interesting content. High-volume publishers can effectively "bury headlines" when a user is provided only a strict chronological view of her data. For example, the microblogging service Twitter provides users with a mostly-chronological view of data; the service quickly found "headline burying" to be a problem, eventually attempting to ban high-volume and manipulative re-tweeting as a type of spam [67].

Our MLS News Reader interface should provide the user with a *snapshot view* of recent activity from among her subscriptions. We intend to limit content shown from each feed in this snapshot, to reduce the problem of “headline burying.”

Subscriptions
The MLS News Reader interface should allow the user to add and remove subscriptions to feeds. Given the somewhat technical nature of exact feed URLs, the application should be able to reasonably auto-discover feeds. For example, the user should be able to enter `cnn.com` as opposed to `http://rss.cnn.com/rss/cnn_topstories.rss`.

Compatibility
As with any modern web application, intentionally designing the service for cross browser compatibility is a necessity. If a user accesses the application from, say, an older version of Internet Explorer, the application should behave similarly to contemporary browsers. Specifically, our application will target compatibility with the set of browsers in Yahoo!’s “Graded Browser Support.” This set of browsers is detailed in Table 2.1.

Browser	*Version*
Internet Explorer	6.0, 7.0, 8.0, 9.0
Firefox	3.*, 4.*, 5.*
Chrome*	Latest stable
Safari	5.*, iOS3.*, iOS 4.*
Webkit	Android 2.*

Table 2.1: Yahoo! Graded Browser Support, as of Jan 2012.
* denotes the most-current non-beta version.

2.5 MLS Feed Sources

To close this chapter, we briefly describe our multilevel feed content, all of which the MLS News Reader should be able to handle. The sources of syndicated web content available on the MYSEA system fall into two categories: (i) sources originating from a single-level network connected to MYSEA, e.g., the Internet and SIPRnet; and (ii) MLS-aware services on MYSEA. The current prototype MYSEA system supports several MLS-aware services that produce syndicated content:

MLS Blogs. Feeds that syndicate recent posts for any user's MLS blog. Currently, the MLS blog software generates feeds in a very flexible manner: it has the ability to publish feeds containing posts at or below the requestor's session level, a subset of those posts about a particular topic, matching a particular search term, etc. The items in these (on-demand generated) feeds are marked with the label of the source documents used during feed generation[1].

MLS Wikis. Feeds that syndicate recent activity for a "web" in the MLS wiki. The organization of the wiki into distinct "webs," each of which is associated with a level, is part of the current wiki software's intended design.

MLS Microblog. Feeds that syndicate all recent activity at a particular level in the MLS microblogging application.

[1]Note: in this example, an untrusted subject is labeling RSS data at the item-granularity. Since the blog application is an untrusted subject, these markings cannot be trusted: as discussed in Section 2.2.1, only a trusted subject can be relied upon to label data at this granularity. Thus, no matter how its contents have been marked, if a *Secret* subject generates a feed, then it is treated as a *Secret* feed (following Section 2.2.2).

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 3: MLS News Reader Prototype Implementation

"Do. Or do not. There is no try."

– Jedi Master Yoda [68, p.187]

We describe our decisions and experiences in prototyping the MLS News Reader application on the current MYSEA system. As previously remarked, the MYSEA system re-uses a high-assurance operating system, currently BAE's STOP OS. We re-use many existing MLS-aware services and support libraries on MYSEA, including Apache and PHP. We begin by explaining our choice of porting an existing feed aggregator.

3.1 Legacy Aggregators in an MLS Setting

For a multilevel environment, reusing existing aggregation software has many benefits. It provides to users in a multilevel context the same features and interfaces they have come to expect, elsewhere. Information flow policy enforcement can be assured by running software without privilege, i.e., as untrusted subjects. When technology changes, as new features become available and new versions of software are released, upgrades can be integrated into the system without its re-certification or re-evaluation. Further, writing aggregation software specifically for an MLS environment, from scratch, would be a costly and non-trivial task. Useful aggregators must be able to discover, parse, sanitize, cache and fix feeds from a variety of sources. Thus, the software must be capable of parsing all popular feed formats, both their current and past versions. Further, it should parse these feeds forgivingly, as feed standards are sometimes not followed faithfully by web publishers.

Sadly, there are relatively few open source, server-side, personal news aggregator projects (see Table 1.1 on page 9). Of those available, most have small user-bases, cannot serve multiple users, i.e., fail to meet our concept of operations, or are commercial services hosted on the Internet (and cannot be privately deployed in a multilevel network or between single-level classified networks). The remaining candidate software cannot be used in MYSEA due to lack of support for some software dependency. For example, the current MYSEA prototype does not fully support Python or JAVA. A further example is MYSEA's current relational database support, which follows a variant of the "distributed, partitioned" MLS Relational Database Management

System (RDBMS) Woods-Hole Architecture [69]. In that architecture, a trusted subject mediates all access to a set of single-level databases, which are isolated from subjects. In MYSEA, databases are isolated outside the MLS LAN on single-level networks, and access is through a trusted database proxy. The proxy does not replicate data from low to high, nor does it simulate a single, trusted MLS RDBMS. Thus, re-using an aggregator that requires PostgreSQL or MySQL database support (i) would require extensive aggregator modification, since it will need to explicitly request access to each single-level database via a proxy, and (ii) would be limited to handling only those data for which there is a corresponding single-level network interface.

Adapting an aggregator to use MYSEA's current MLS RDBMS would cause us to lose most of the benefits associated with software reuse: heavy modifications to the software would preclude rapidly integrating improvements to the software from other sectors. Further, we would lose scalability, because we could only support as many levels as we have single-level networks. Given that MYSEA hosts services producing feeds at arbitrary levels, we decided that MYSEA's current "LAMP stack" was unable to support *any* surveyed project (in a manner that could satisfy our requirements).

These factors led to our decision to design and develop a new web application for our MLS News Reader service. In our prototype implementation, the PHP library SimplePie [70] provides the logic for retrieving, parsing, and caching feeds. SimplePie was selected as (i) it was used by many of the projects we surveyed, (ii) it appeared to have fair community support and documentation, and (iii) was relatively straightforward to port to MYSEA. Thus, our prototype implementation *does* attempt to re-use, when possible, existing libraries and software. The MLS News Reader's design, however, does not make use of "legacy aggregators" in a black-box manner.

3.2 Porting SimplePie

To verify that SimplePie could be used to support our design, two types of preliminary testing were done: (i) verifying SimplePie would work on STOP OS, by installing and running its compatibility and unit tests, (ii) verifying that SimplePie caching worked in read-only mode, even from levels with no network access. We validated the latter feature by running a sample application using SimplePie on Linux, filling the feed cache, modifying the cache to be read-only, disconnecting the network, and trying to access those feeds. Both tests were successful.

It has been the case in the past that limiting an application to read-only access (when it was developed with the anticipation of having full read-write access), sometimes leads to a variety of hard failures or other anomalies. Indeed, errors are displayed when the SimplePie library attempts to update a cache when it lacks permissions. In our prototype, using permissions introspection, the application assesses if it has sufficient permissions to update a cache. When it lacks these permissions, it voluntarily sets its "access mode" to have a very long cache timeout, so it does not attempt to update the cache. This is, of course, merely to avoid logging errors and not for policy enforcement. Similar logic is used to detect when image and favicon cache-misses should be ignored, or corrected.

3.3 Project Structure

The key components of our MLS News Reader application include the SimplePie library, the application logic, support libraries, style files and cache (see Figure 3.1). The single-level state slices are stored under each user's home directory.

The SimplePie library is wholly contained within the `simplepie.inc` file. Our main file, `index.php`, uses PHP's `include` function to access logic to construct various "views" for the user-interface: preferences view (`in-prefs.php`), chronological view (`in-chrono.php`), chronological view within a level (`in-chrono_level.php`) and single feed view (`in-list.php`).

3.4 Cache and Preferences

There is a single cache per level, shared amongst all users. Thus, the cache can be updated by any MYSEA user at the appropriate level. As in Figure 3.1, the location of each feed cache is:

```
[document root]/news/cache/[security level]
```

The idea of implementing separate caches for each user was rejected as inefficient, despite it being arguably favorable from a DAC perspective. For example, the MLS wiki engine in MYSEA produces automatic feeds for all pages, including those with wiki access controls. The design of MYSEA causes the MLS News Reader to run with the identity of the remote user and the wiki automatically authorizes requests to restricted pages based on this identity. This opens the possibility of the MLS News Reader accessing private feeds, and storing this data in the public cache. Of course, DAC always provides the ability to release private data to a public venue. A future enhancement might include a "private" toggle during subscription, to provide the user with the option of using a private cache or disable caching altogether, on a feed-by-feed basis.

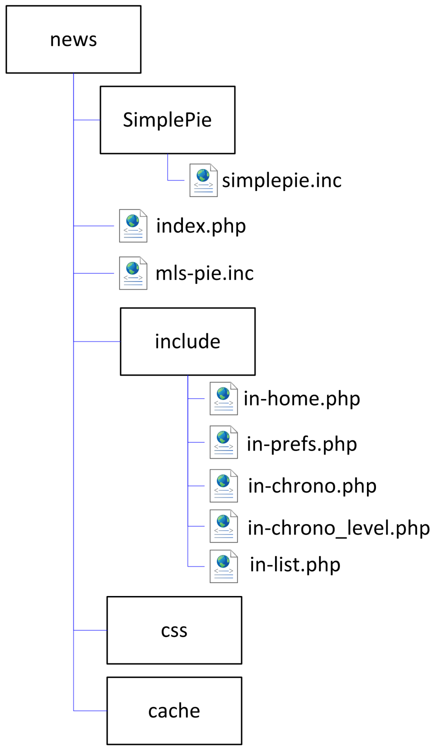


Figure 3.1: MLS News Reader directory structure (abbreviated).

3.4.1 User State Slices

Since the MLS News Reader is, for each user request, launched with the identity of the remote user, the process has access to all files and directories accessible to that user. To prevent one user from accessing the state of another user, the application stores user state in a private location, under her home directory:

```
/home/[username]/[security level]/.rss/config/default
```

When a user accesses the preference view (see Figure 3.2), she is given the option of adding or removing feeds from the state slice associated with her *current* session level. After submitting this form, the application verifies the remote feed as accessible, creates an entry for the feed, and serializes this entry as state. The user is also presented with a list of those subscriptions that she may choose to delete. In Section 3.5, we discuss the motivations related to limiting subscription to those feeds accessible at the user's session level.

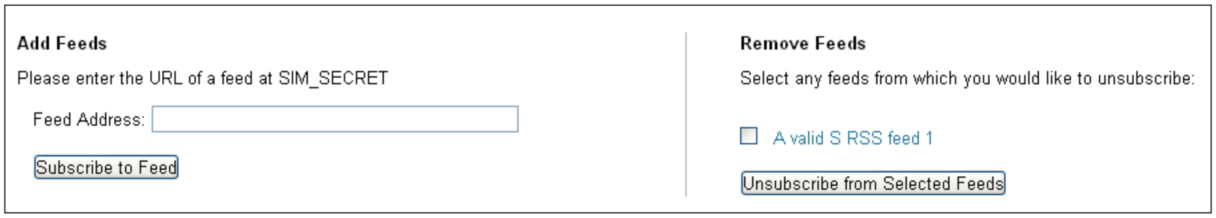

Add Feeds
Please enter the URL of a feed at SIM_SECRET
Feed Address:
Subscribe to Feed
Remove Feeds
Select any feeds from which you would like to unsubscribe:
A valid S RSS feed 1
Unsubscribe from Selected Feeds

Figure 3.2: MLS News Reader Preferences Add and Delete Forms.

3.5 Subscribe-Down

It is reasonable to consider the ability to subscribe to feeds at *any* level *dominated* by the user's session level. We call this *subscribe-down*. For example, from an information flow policy perspective, a user at *high* could read a *low* feed from the cache, regardless of the subscriptions in the user's *low* state slice. One possible interpretation of such a feature is that, while the feed is at *low*, the fact that the user has subscribed to it is, itself, classified at *high*.

In our design, however, subscribe-down gives rise to a pathological scenario in which (say) *Secret* users are subscribed to an *Unclassified* feed, to which no *Unclassified* users are subscribed. Thus, the *Unclassified* cache will never be updated. For *Unclassified* feeds produced by multilevel services, this poses no problems: a *Secret* user can request the *Unclassified* feed from the multilevel service directly, without caching. For feeds whose source is a single-level network service, however, caching is integral. For example, if a user has subscribed to a feed on the Internet while at *Secret*, it may be the case that this feed is never cached.

This pathological scenario for subscribe-down can be ameliorated. For example, a downgrading service could retrieve the *Unclassified* feed names from any *Secret* subscription list, and pass this to the *Unclassified* cache update service; however, since the cache update service is a single-level subject, this would open a signaling channel. More generally, updating a *low* cache based on information at *high* will ultimately defeat whatever point is intended by subscribing to a feed at *high* rather than at *low*. Thus, we opted for a simpler design, free of channels, that does not support subscribe-down.

3.6 Input Validation

Whenever an application takes user input, it is important that it be validated. This is necessary in order to keep the user interface sensible and to avoid security threats [71, pp.174–176]. The MLS News Reader contains only one form for user input, on the preferences page (Figure 3.2).

Validation logic checks that the user input is non-empty, at which point the feed passes through SimplePie's feed sanitization logic. In particular, SimplePie strips HTML tags (e.g., `blink`, `font`, `marquee`) and certain HTML attributes (e.g., `bgsound`, `onmouseover`, `onerror`). Regardless, the creators of SimplePie provide the disclaimer,

> "If you don't trust the feeds that you're parsing, you should do your own data sanitization to avoid security issues. If you DO trust the feeds you're parsing, this shouldn't be an issue [72]."

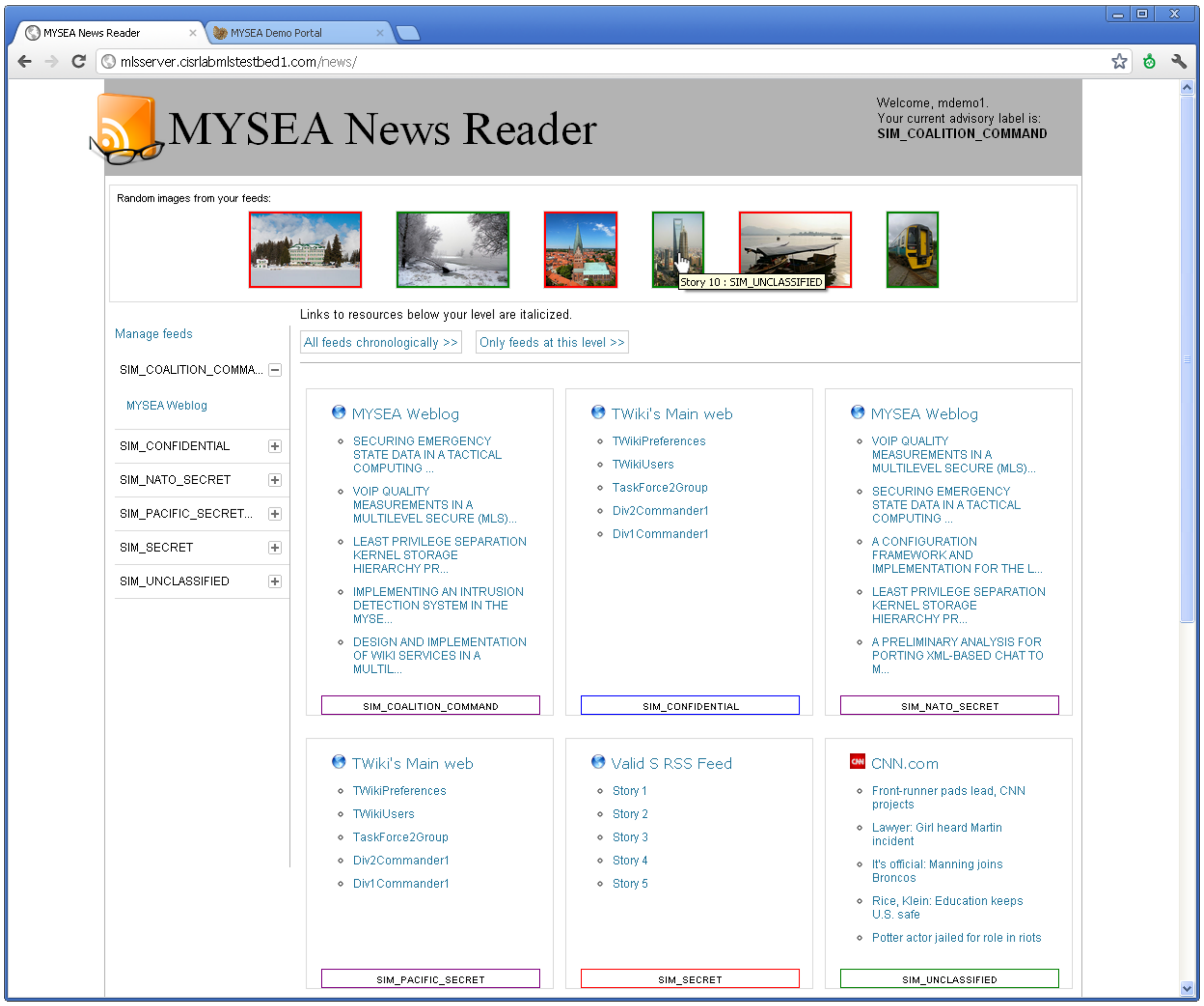
By design, our *whole* application is considered untrusted and is effectively sandboxed under the user's identity. Thus, the need to sanitize input strictly is largely obviated: a user's malicious input can do no greater harm than the user's permissions already permit.

3.7 User Interface

As per our requirements, the aggregator displays data carrying different security labels and we must communicate these classifications to the user, as advisory labels. User interfaces for digital data carrying (possibly) hundreds of different labels have not been sufficiently explored in the literature, or described by regulation. Most regulations fail to describe marking documents derived from multiple sources, for anything beyond physical documents with very few derivative labels [73–75]. It is customary to mark *digital* media following physical document-marking regulations, in which parenthetical abbreviations are used to mark the paragraph- and section-level document structure.

RSS and Atom, by design, separate document structure (XML) from its presentation (handled by a style sheet or XSLT transform). The convention that user interfaces for data should largely reflect the structure and format of its source is, at best, naïve, especially when considering digital publishing. In particular, using parenthetical abbreviations for each subsection fails to allow rich dynamic content, interfaces that lack section structures, or obscure labels for which no parenthetical abbreviation system can be adopted. We approach this problem from several angles in our application.

Each summary box on our main page contains a footer with a color-bordered security label (see Figure 3.4). The color-coding is customizable by the system administrator; following the coloration of standard DoD forms (Figure 1.1 on page 2), we chose green for *Unclassified*,


MYSEA News Reader | MYSEA Demo Portal

mlsserver.cisrlabmlstestbed1.com/news/

MYSEA News Reader

Welcome, mdemo1.
Your current advisory label is:
SIM_COALITION_COMMAND

Random images from your feeds:

Story 10 : SIM_UNCLASSIFIED

Links to resources below your level are italicized.

Manage feeds

SIM_COALITION_COMMA...
- MYSEA Weblog

SIM_CONFIDENTIAL
SIM_NATO_SECRET
SIM_PACIFIC_SECRET...
SIM_SECRET
SIM_UNCLASSIFIED

All feeds chronologically >> | Only feeds at this level >>

MYSEA Weblog
- SECURING EMERGENCY STATE DATA IN A TACTICAL COMPUTING ...
- VOIP QUALITY MEASUREMENTS IN A MULTILEVEL SECURE (MLS)...
- LEAST PRIVILEGE SEPARATION KERNEL STORAGE HIERARCHY PR...
- IMPLEMENTING AN INTRUSION DETECTION SYSTEM IN THE MYSE...
- DESIGN AND IMPLEMENTATION OF WIKI SERVICES IN A MULTIL...

SIM_COALITION_COMMAND

TWiki's Main web
- TWikiPreferences
- TWikiUsers
- TaskForce2Group
- Div2Commander1
- Div1Commander1

SIM_CONFIDENTIAL

MYSEA Weblog
- VOIP QUALITY MEASUREMENTS IN A MULTILEVEL SECURE (MLS)...
- SECURING EMERGENCY STATE DATA IN A TACTICAL COMPUTING ...
- A CONFIGURATION FRAMEWORK AND IMPLEMENTATION FOR THE L...
- LEAST PRIVILEGE SEPARATION KERNEL STORAGE HIERARCHY PR...
- A PRELIMINARY ANALYSIS FOR PORTING XML-BASED CHAT TO M...

SIM_NATO_SECRET

TWiki's Main web
- TWikiPreferences
- TWikiUsers
- TaskForce2Group
- Div2Commander1
- Div1Commander1

SIM_PACIFIC_SECRET

Valid S RSS Feed
- Story 1
- Story 2
- Story 3
- Story 4
- Story 5

SIM_SECRET

CNN.com
- Front-runner pads lead, CNN projects
- Lawyer: Girl heard Martin incident
- It's official: Manning joins Broncos
- Rice, Klein: Education keeps U.S. safe
- Potter actor jailed for role in riots

SIM_UNCLASSIFIED

Figure 3.3: The MLS News Reader index page.

blue for *Confidential*, red for *Secret* and orange for *Top Secret*. Currently, all other levels and categories receive the same a default color, purple.

A random image banner (Figure 3.5) labels information in the same way, while providing links to relevant news items. Each image is bordered by the label color for the feed from which it originated. Additionally, when the user's mouse hovers over the image, a tool tip displays its security label and feed name.

The index page is organized to most prominently display data associated with the user's session level. For example, if the user is logged in at *Secret*, the summary boxes are ordered such that *Secret* summary boxes appear before *Unclassified* summaries. Further, the left-hand-side

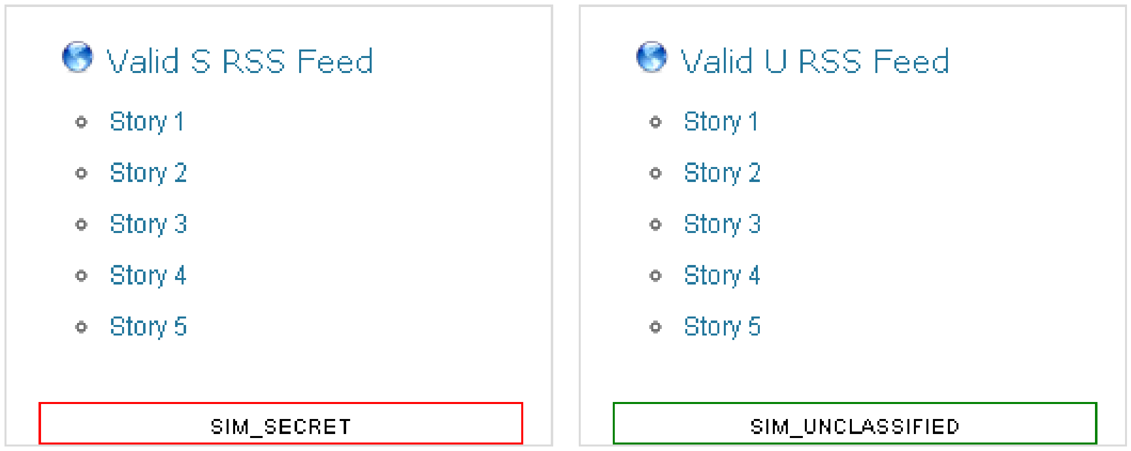


Figure 3.4: Summary boxes, color-coded by security labels.

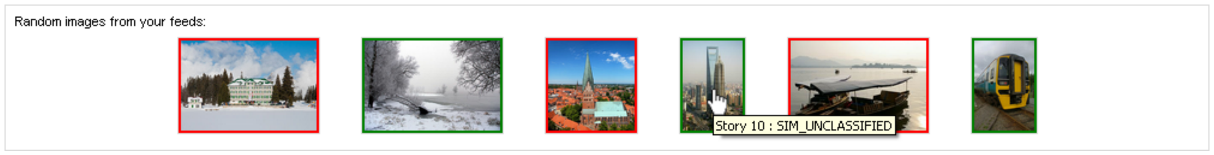


Figure 3.5: Random image banner, color-coded by security label.

navigation auto-opens an accordion list of subscriptions for the current session level, while the others are closed by default (Figure 3.6 and Figure 3.7). This is intentional for two reasons: (i) the user is presented with information that is likely most relevant to her, based on her session level, and (ii) auto-focusing the current level and de-emphasizing others de-clutters the work environment, when working across many levels.

3.8 Implementation Challenges

The dynamic nature of variable subscriptions and variable feed content added significant complexity to the implementation of the MLS News Reader. For static content, a web designer may adjust font size and truncate text as needed. When all content is known and static, image handling is relatively straight-forward. For dynamic content, however, complications arise. The web designer has little control over the quantity or quality of information fed to the application. For example, the aggregator must be able to elegantly handle zero feed subscriptions, or hundreds of feeds at hundreds of levels.

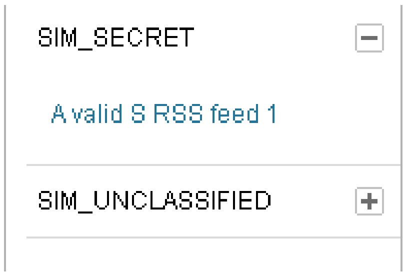


Figure 3.6: Default navigation accordion for a *SIM_SECRET* session.

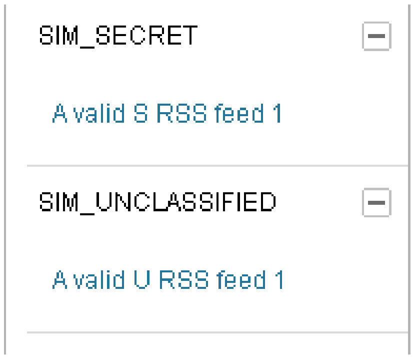


Figure 3.7: Navigation accordions, opened.

A non-exhaustive list of some issues we encountered includes:

- Feeds with no items;
- Feeds with no title;
- Feed titles containing odd characters, all capitals, no spaces (or a combination thereof);
- Feeds with very long descriptions (e.g., over 500 characters);
- Feeds containing tracking logic (requests to 1 pixel GIFs, used to monitor behavior)

Using various exception handling tests, we validated that our implementation handles all of the above, in an acceptable manner (see Section 3.9).

PHP Limitations

The PHP package supported by MYSEA lacks the GD image processing library and the Alternative PHP Cache (APC). The lack of GD means, in particular, that PHP cannot be used to resize images. As a result, it was necessary to resize images for the image banner using HTML and cascading style sheets (CSS). The lack of APC results in less than optimal performance, upon which we elaborate in Section 5.5.

“Bad” Images

We observed that major online publishers tend to include tracker pixels with their feeds and feed items. This may be accomplished using feed advertising software, such as *feedburner* and *pheedo*. For example, *pheedo* claims to “turn your RSS traffic into money” [76]. Sadly, this wreaks havoc for feed aggregators attempting to extract and display embedded images. Given that RSS does not provide a clean method of embedding images, it is necessary to *scrape* images from the description field at the item level. The MLS News Reader fetches these items from the cache, so embedded image references like `<img src="example.jpg">` instead resemble:

```
<img src="mlsserver.cisrlabmlstestbed1.com/news/handler_image_
SIM_UNCLASSIFIED.php?i=e39e0c2017f9c7e823494bzffaaffe311">
```

In other words, the caching logic obscures the image extension. Instead, we scrape the feed using regular expressions, isolating the `<img src="...">` reference, and fetch the image using the `getimagesize` function in PHP. This requires an HTTP `GET` request which, on average, took roughly .3 seconds per image when fetching from the cache (measured using the `microtime` function in PHP). As this operation seemed rather costly, we decided to limit the number of `GET`s incurred by the random image banner logic (see Appendix D for a sample of this logic).

3.9 Prototype Testing

We designed the MLS News Reader to handle a variety of feed content, on multiple web browsers. We developed a test plan, designed to assess if the MLS News Reader satisfies reasonable functional and exception requirements. In particular, our tests are designed to validate:

- Users are able to subscribe to a feed at *low* and read its contents at *high*. Additionally, feeds at *high* are unavailable at *low*.
- If an element is displayed outside the context of its source feed, it is marked with the correct label. For example, items in a feed (e.g., while in chronological view) and images from a feed (e.g., in the random image banner) are correctly labeled.
- Feeds are labeled correctly and are properly listed in the navigation bar and summary boxes. This includes validating that unusual text combinations (e.g., long strings of capital letters without spaces) are not formatted strangely for the user (i.e., long names are truncated with ellipses, names appear in tool tips properly).
- The preferences view and its forms behave similarly in different browsers. For example, on Mozilla Firefox and Google Chrome, the "Enter" key can be used to submit the form, while on Internet Explorer it was necessary to tweak the form with a hidden element in order to obtain the same functionality.

3.9.1 Procedure and Results

The test procedure is provided in Appendix A. The procedure validates the functional properties of third-party dependencies (i.e., SimplePie) by leveraging existing compatibility tests. Our prototype MLS News Reader passed all functional and exception tests.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 4: MLS News Reader Prototype Performance Evaluation

"We affirm that the beauty of the world has been enriched by a new form of beauty: the beauty of speed."

–Filippo Tommaso Marinetti [77, p.51]

Our MLS News Reader design precludes the use of several efficient web application practices. For example, we split user data into a series of flat files and we invoke library routines in artificial ways to access multiple feed caches. It would almost certainly be faster to keep information in a single in-memory cache or leverage a single, fast database query engine for all data; however, such a restructuring would likely require that our web application become trusted. We also inherit several design choices from MYSEA: the MLS web server spawns an untrusted process to service each connection, all network access is performed through a trusted MYSEA socket proxy, etc. In this chapter, we analyze the performance of our design, especially trends related to scaling across feeds, items and levels.

4.1 Test Strategy

We elected to use Apache JMeter [78] to measure HTTP response and data transfer times, under a variety of scenarios. For a complete description of our experiment set-up and scenarios, see Appendix B. Performance test scenarios include:

- Loading feeds at a single level, i.e., at *Unclassified*.
- Loading feeds at multiple levels, i.e., performing read-down.
- Loading feeds with images, exercising our image banner logic.

We are particularly interested in the average case performance experienced by the end-user, so the feed cache is pre-populated for all of our tests. Our performance does not reflect the extra fetching logic that occurs on a cache timeout. Following our design, this case can be handled by a cache update daemon rather than impact the performance experienced by end users.

All labels and data are simulated, rather than authentic, classified data (thus, *SIM_SECRET* rather than *Secret*). The system is connected to three single-level networks: the Internet and two (simulated) classified networks. At all other session levels, no other networks are available;

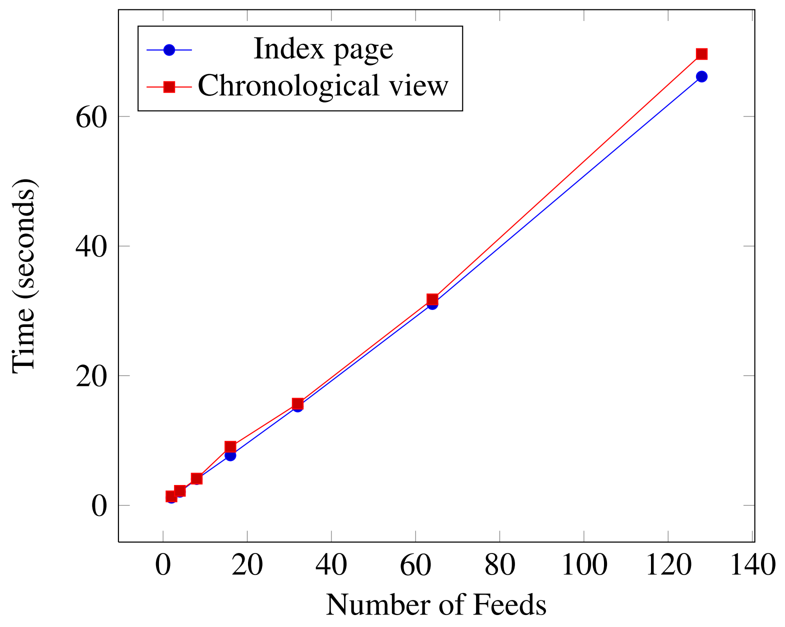


Figure 4.1: Loading time vs. number of feeds, at a single level (for data, see Table 4.1 and Table 4.2).

however, at any level, numerous MLS applications on MYSEA generate feeds. Developing our experiment infrastructure required automatically generating various feeds and subscriptions (i.e., `default` preference files) at each level (see Appendix C). We made no attempt to simulate "realistic" user loads or to simulate "realistic" feed data or feed sizes.

4.2 Analysis

We summarize the performance trends we observed in our preliminary analysis. Experiments were repeated 20 times and observations appeared highly normal; we believe the mean behavior accurately reflects the major performance trends. We emphasize that our experiments were performed in a virtualized testbed (rather than on bare metal equipment), using the most recent version of the prototype MYSEA system (version 5). Thus, it is not fair to draw conclusions related to absolute performance costs, and we only attempt to characterize performance trends.

4.2.1 Scaling across feeds

We observe that there is near-constant overhead associated with processing feeds and items. Varying the number of feeds demonstrates very linear behavior (see Figure 4.1). Expectedly, the cost of loading the index page when feeds have images is marginally greater (~40ms per feed, in our tests). We further discuss the costs associated with images and the image banner in Section 4.2.3.

No. of feeds	Avg time (ms)	Data transferred (kb)	Throughput (req./min.)
2	1176	1.3	51.0
4	2093	2.0	28.7
8	4049	3.5	14.8
32	15248	12.3	3.9
64	31052	24.1	1.9
128	66151	47.6	0.96

Table 4.1: Index page with text feeds at one level (Test 1, Appendix B).

No. of feeds	Avg time (ms)	Data transferred (kb)	Bandwidth (req./min.)
2	1385	20	43.29
4	2239	34.7	26.79
8	4124	64.1	14.56
16	9042	123	6.63
32	15686	241.6	3.82
64	31779	478	1.89
128	69629	952.3	.8617

Table 4.2: Chronological view with text feeds at one level (Test 5, Appendix B).

No. of feeds	Avg time (ms)	Data transferred (kb)	Throughput (req./min.)	Images
2	5268	16	11.39	6
4	6002	23	9.99	6
8	8121	38	7.34	6
16	12465	67.3	4.81	6
32	21219	126	2.83	6
64	38700	243.5	1.55	6
128	76027	478.5	.7892	6

Table 4.3: Index page with image feeds (Test 2, Appendix B).

4.2.2 Scaling across levels

While reading down and combining feeds at multiple levels, we again see a clear linear trend when dealing with more feeds and items (see Figure 4.2 and Figure 4.3). Interestingly, reading down appears much faster than reading in a single-level context. In some situations, reading to two caches takes half the time: for the index page, ~51ms per-feed (single-level) vs. ~27ms per-feed (multilevel).

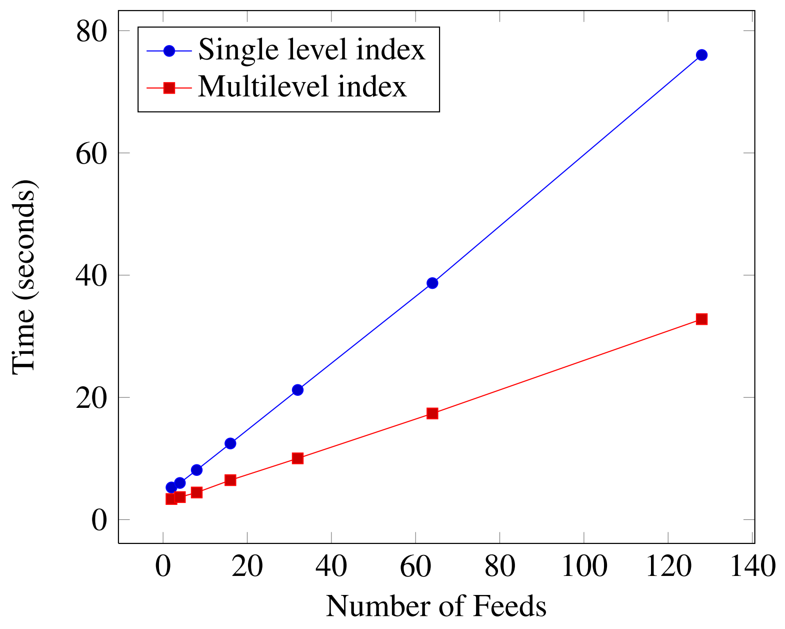


Figure 4.2: Index with images, single level vs. multilevel (for data, see Table 4.3, and Table 4.4).

No. of feeds	Avg time (ms)	Data transferred (kb)	Throughput (req./min.)	Images
2	3377	16620.5	17.76	5.8
4	3689	23986.7	16.26	6
8	4443	38498.0	13.50	6
16	6462	67608.1	9.28	6
32	10028	125912.1	5.98	6
64	17374	242651.4	3.45	6
128	32802	476053.2	1.83	6

Table 4.4: Multilevel index page with images (Test 4, Appendix B).

This apparent paradox has a relatively simple explanation. Our single-level experiments are at *Unclassified*. Despite the fact that the experiment is designed to pull content exclusively from cache, the SimplePie library makes a `gethostby` request per feed. At *Unclassified*, this call causes a DNS query to the Internet; at all other levels, the query either fails (the MYSEA resolver library immediately responds as if a timeout has occurred) or is answered by a local DNS server on a simulated classified network. Thus, processing *Unclassified* feeds while at *Unclassified* is slightly more expensive than processing the same feeds at a higher level, due to network effects.

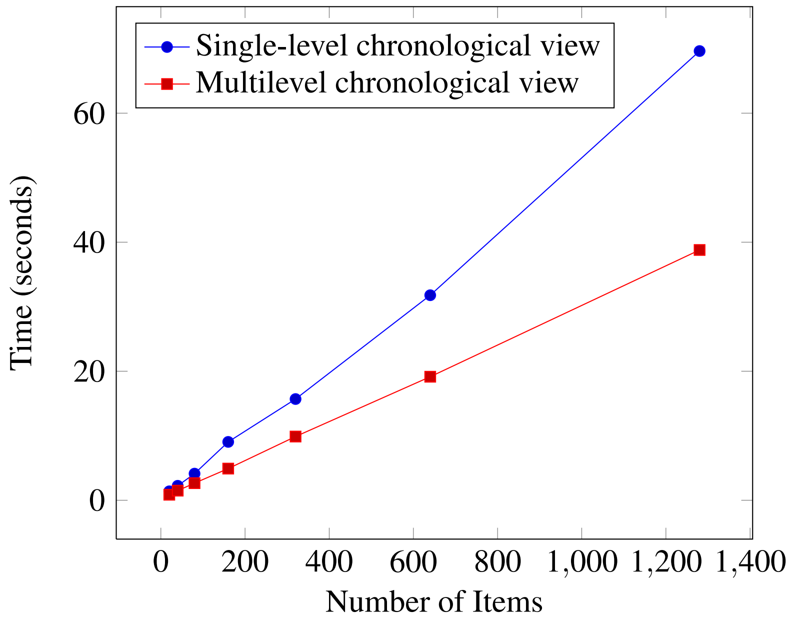


Figure 4.3: Loading time vs. number of feeds, reading one and two levels (for data, see Table 4.2, and Table 4.5).

No. of feeds	Avg Time (ms)	Data Transferred (kb)	Throughput (req./min.)
2	853	2.1	70.31
4	1487	3.6	40.32
8	2649	6.4	22.64
16	4905	12.3	12.23
32	9873	24	6.08
64	19144	47.5	3.13
128	38791	94.5	1.55

Table 4.5: Multilevel chronological view, no images (Test 6, Appendix B).

We were interested in determining if additional MAC checks by the operating system, or reading from two different caches, resulted in a system performance bottleneck as we dealt with more and more feed data. It certainly appears to be no more of a performance bottleneck than in the single level case. It is likely that all MAC checks and cache requests are equal in cost, and network topology has a larger impact on performance at higher levels.

4.2.3 Image Banner Performance

The random image banner, while a relatively minor design feature of our aggregator, posed several significant implementation challenges during prototype construction, e.g., costs associated

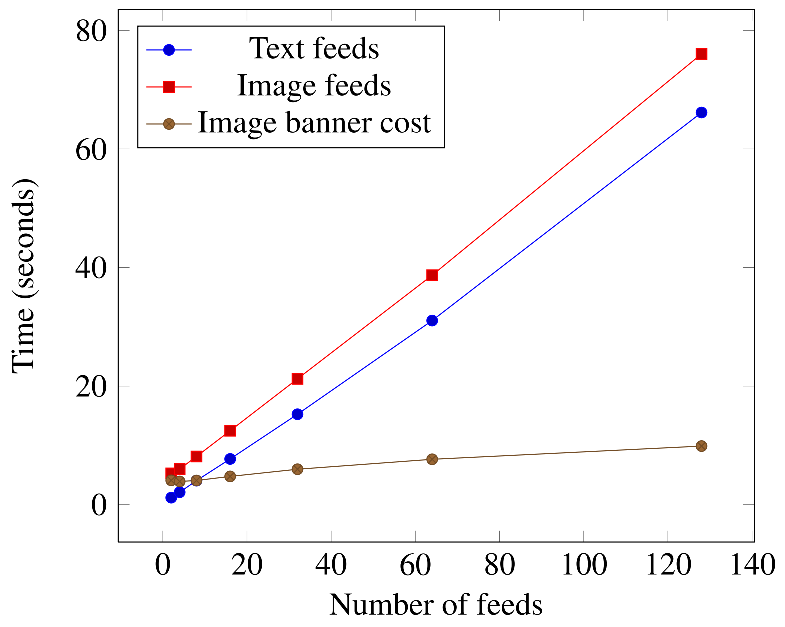


Figure 4.4: Index with text feeds vs "good" images, at single level (for data, see Table 4.1, and Table 4.3).

No. of feeds	Avg time (ms)	Data transferred (kb)	Throughput (req./min.)	Images
2	2937	13.7	20.42	1
4	5038	21.4	11.91	2
8	8940	36.6	6.71	3.7
16	14113	66.4	4.25	3.3
32	23251	125.1	2.58	4.2
64	42423	242.4	1.41	3.7
128	80471	477.2	.7456	4.5

Table 4.6: Index page with mixed "good" and "bad" images, at one level (Test 3, Appendix B).

with searching for images in feeds and filtering-out web bugs. For feeds with many images, this overhead proved to be costly. A limit was placed on the number of images processed during banner construction, i.e., the first six appropriate images would be used else the logic terminated when a limit is reached.

Under optimal conditions, this as a near-constant overhead: ~5760ms for six images (Figure 4.4), or ~960ms per image. This overhead includes searching feed items for image references, performing a `GET` on the image, and transferring the image to the client. When some feeds hold images that must be filtered, i.e., some "bad" images, we see this cost jump to

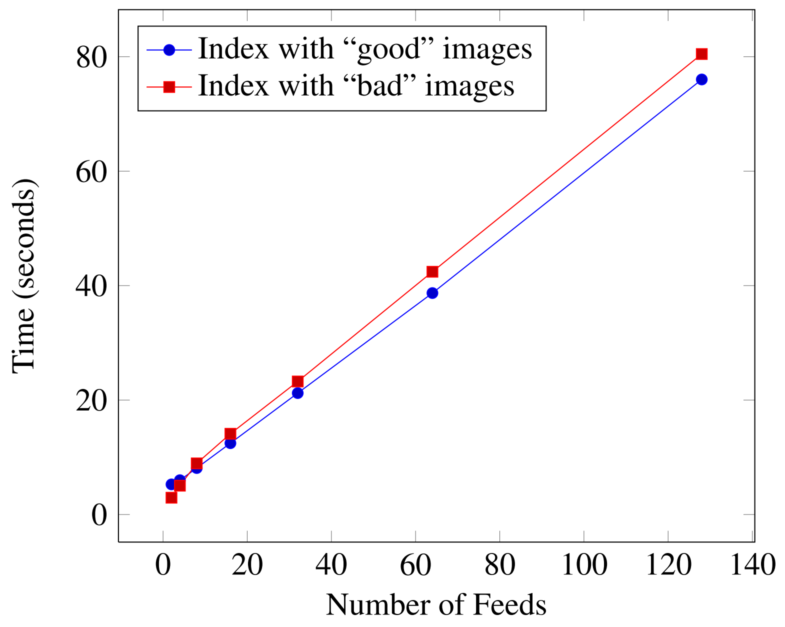


Figure 4.5: Index with “good” images vs. index with “bad” images. Both single level. (For data, see Table 4.3 and Table 4.6).

~7100ms (Figure 4.5), an increase of ~20%. This reflects the fact that “bad” images are smaller (i.e., like tracking GIFs) and less costly to process, but is most reflective of our test mixture—in our “mixed” test feeds, with high probability we will process one or two extra images, but it is very unlikely we will need to process six or more “bad” images. Again, we do not attempt to simulate the content of a “typical” feed; for some feeds, e.g., `cnn.com`, tracking pixels are much more common than “good” images.

Certainly, a user subscribed to 100 feeds is not going to want to wait 50 seconds for a page to load. Pagination is one promising method to limiting the content processed for the index and chronological views, improving load time. In Chapter 5, we discuss other methods of improving performance.

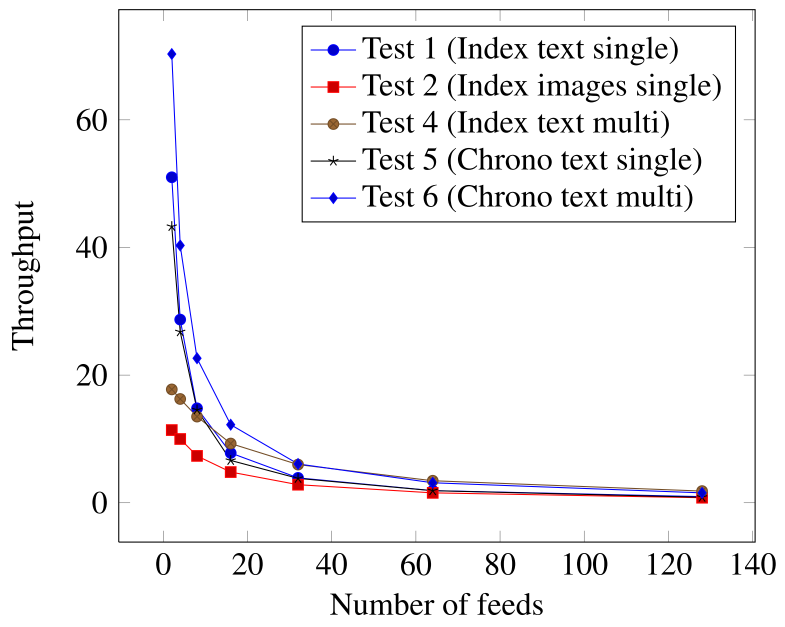


Figure 4.6: Throughput (requests per minute) for each test.

4.2.4 Throughput

For all of our tests, we see relatively similar throughput curves (Figure 4.6). The more subscriptions (e.g., 128 feeds), the fewer requests can be served, per unit of time. The primary conclusion is that our application is not bandwidth-limited, but is limited by server-side logic. Again, pagination and view limits have been removed, artificially, so that all views show as many feeds as possible. If we were to limit the number of feeds and items displayed, we could use estimations based on throughput observed in a production multilevel LAN to maximize average load time. From our observations here, limiting views to no more than ~20 feeds, or no more than ~200 items, could significantly improve application performance.

CHAPTER 5: Future Work

"The greatest enemy of a good plan, is the dream of a perfect plan"

– Carl von Clausewitz [79]

We discuss some promising strategies which may improve the functionality and performance of our MLS News Reader prototype.

5.1 State slices supporting complex queries

A flat file database, such as SQLite, could allow us to manage more complex user state information. For example, it would be possible to mark certain articles as "read" or to tag feeds with keywords. The database could also hold user-customization data. For example, a user could choose to always have `cnn.com` in her first summary box on the index page, regardless of level.

5.2 Text Scraping

Currently, resources referenced by a feed (but not *in* the feed) are unavailable at higher levels, since that content must be fetched from some single-level network at a level other than the user's session. The MLS News Reader caches content contained within the `description` tag, but does not fetch the linked article to cache *its* content. Applications like Instapaper [80] demonstrate the convenience of scraping text in online articles and storing them for offline use. It may be challenging for such an application to distinguish between main content and extraneous information (e.g., advertising, links). It remains to be seen how web publishers will handle text scrapers as their popularities increase. Instapaper allows publishers to opt out, but claims that no major publishers have yet chosen to do so [81]. In any event, incorporating this type of technology could greatly increase the quantity of information served by the MLS News Reader.

5.3 MYSEA Service Integration

We can envision methods of better facilitating information sharing amongst MYSEA applications. For instance, a user could click a link on an item within the MLS News Reader, to post a link to the article on the MYSEA microblogging service. This gives the user a simple means of sharing content with her microblog "followers." Such "sharing" features already exist in popular services, such as `cnn.com` and `amazon.com`. Increasing the ease with which sharing takes place among the MYSEA web applications increases the utility of each.

5.4 Re-Aggregation

As mentioned in Section 1.4.5, some aggregators allow a user to re-publish customized feeds. This can take the form of chronological re-publishing of all feed items, or some selection of items from the feed. Supporting this feature would present new issues in a multilevel environment. For instance, should a user be allowed to mix different levels of information and re-publish them at the highest level? Or, should the user be restricted to publishing a per-level feed?

5.5 Performance Enhancements

There are several avenues worth exploring, to improve service performance without re-designing the MLS News Reader application or the platform on which it runs.

Cache Maintenance.
SimplePie does no maintenance of its cache files. If its cache folder grows too large, this could degrade performance. In addition to the *trusted cron service* proposed in Section 2.4, it would be felicitous to create a single-level cron job to periodically remove older feed cache entries.

Port PHP Enhancements to MYSEA.
An opcode cache in PHP, known as APC, has been shown to increase requests per second and shorten request time [71, pp.210–215] [82, pp.41–42]. APC is not currently available on MYSEA's PHP implementation. Other useful PHP caching mechanisms not currently available are memcache and `tmpfile()`. The GD Image Library, also missing in MYSEA's PHP, is the standard PHP method of resizing images. Porting this, and leveraging an SQLite database to mark processed images and store resized images, could improve the random image banner processing logic.

Client-side logic.
Work was started on a JavaScript-based “overlay” which could (quickly) provide a user with more items from a particular feed. This “quick view” would speed up the ability of the user to peruse information without cluttering the interface. Similarly, AJAX-powered feed limiting could limit the up-front costs associated with page loading. Extra feed items could be fetched asynchronously, after a page has loaded.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 6: Conclusions

“Dreams are not what you see in sleep. They are the things that do not let you sleep.”

–Dr. A.P.J. Abdul Kalam [83]

This thesis explored the relationship of dynamic web content sharing and mandatory information flow policies, in the context of MLS web syndication. We have described several concepts in aggregating labeled web content, and demonstrated that they can be implemented with few trusted components. To our knowledge, managing labeled, syndicated web content has not been previously explored in a multilevel context. In particular, managing web content labeled with more than a few labels has been largely ignored, due to the lack of commercial architectures supporting remote user sessions at arbitrary levels. We validated the concept of multilevel web syndication by implementing and evaluating a prototype MLS News Reader application, running on MYSEA, written in PHP and using the popular SimplePie library. Our MLS News Reader prototype demonstrates the plausibility, utility and complexity of MLS web syndication. In particular, we shared insights related to the challenges of presenting multilevel information in a useful and user-friendly manner.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A:
MLS News Reader Functional and Exception Test Plan

A.1 References

The following is a list of references required to complete the Test Procedure:

1. MLS News Reader Test Plan.
2. Current version of MYSEA code.

A.2 Description

This Test Procedure is intended to implement the MLS News Reader Test Plan.

A.3 Test Setup

This Test Procedure requires the following setup procedures to have occurred:

1. MYSEA Server Setup.

A.3.1 MYSEA Server Setup

The standard MYSEA demo server setup is used.

A.4 Abbreviations

The following abbreviations are used in this document:

`MYSEA_WEBSERVER`	`http://mlsserver.cisrlabmlstestbed1.com`

A.5 Functional and Exception Testing Coverage

Specify the browser and version number used in your testing.

- The web browser used during this test:____

This test procedure should be repeated once per browser. Minimally, test (some version of) Mozilla Firefox and (some version of) Internet Explorer.

A.6 MLS News Reader Functional Test Details

For each test:

- TPE 1 is associated with Client 1[2].

A.6.1 Functional Test: SimplePie Compatibility Test

Test Instruction

1. Establish a session at SIM_UNCLASSIFIED for mdemo1 using TPE 1.
2. Start a web browser on Client 1.
3. Navigate to the URL:
   `http://MYSEA_WEBSERVER/news/simplepie/compatibility_test/sp_compatibility_test.php`
4. The status should indicate that all dependencies are satisfied to use the SimplePie library; in particular, the expected result is the following:

Other libraries may be enabled. The final summary message should read:

1. Establish a session at SIM_UNCLASSIFIED for mdemo1 using TPE 1.
2. Start a web browser on Client 1.
3. Navigate to the URL:
   `http://MYSEA_WEBSERVER/news/simplepie/compatibility_test/sp_compatibility_test.php`
4. The status should indicate that all dependencies are satisfied to use the SimplePie library; in particular, the expected result is the following:
   4.a. PHP version meets requirements
   4.b. XML enabled, sane
   4.c. PCRE enabled
   4.d. mbstring enabled
   4.e. iconv enabled
   Other libraries may be enabled. The final summary message should read:

Bottom Line: Yes, you can!

Your webhost has its act together!

[2]Please note this is adapted to work with our prototype system which had one client and thus one TPE. Furthermore, given that our prototype system did not contain the current version of the blogging software or the microblogging service, associated test are also omitted.

5. Close the web browser.
6. Logout TPE 1.

A.6.2 Functional Test: SimplePie Automated Tests

Test Instruction

1. Establish a session at SIM_UNCLASSIFIED for mdemo1 using TPE 1.
2. Start a web browser on Client 1.
3. Navigate to the URL:
   `http://MYSEA_WEBSERVER/news/simplepie/test/test.php`.
4. The expected result is to pass at least 97% of the automated tests. The following errors are acceptable (validated on Linux):
   4.a. Feed Title: 96% passed (2/52 tests failed)
   4.b. First Item Title: 81% passed (18/95 tests failed)
   4.c. iTunesRSS: 0% (3/3 tests failed)
5. Close the web browser.
6. Logout TPE 1.

A.6.3 Functional Test: Test feed detection and feed parsing

This tests auto-detection of feeds in MYSEA, and that SimplePie can recognize and parse various demo feeds. These are as much tests of SimplePie as they are of the feeds on which the MLS NEWS READER demo relies.

Test Instruction

1. Establish a session at SIM_UNCLASSIFIED for mdemo1 using TPE 1.
2. Start a web browser on Client 1.
3. Navigate to the URL:
   `http://MYSEA_WEBSERVER/news/simplepie/demo/`
4. Input feeds at the following addresses
   4.a. `cnn.com`
   4.b. `http://MYSEA_WEBSERVER/twiki/bin/view/Main`
5. Verify that, for each, a feed title and feed items are listed.
6. Close the web browser.
7. Logout TPE 1.

A.6.4 Functional Test: Add a feed, as a new user

This test assumes user cudemo has no subscriptions. This can be assured using the command `rm -rf /home/cudemo/*/.rss` executed by `madmin` on the MYSEA server hosting the News Service. The current demo setup should, however, be in this configuration already (*i.e.*, `cudemo` has no pre-registered subscriptions in the demo).

Test Instruction

1. Establish a session at SIM_SECRET for cudemo using TPE 1.
2. Start a web browser on Client 1.
3. Navigate to the URL `http://MYSEA_WEBSERVER/news`
4. The expected result is the MLS NEWS READER main page, with no feeds showing.
5. Click 'Manage Feeds.'
6. Input the address:
   `http://MYSEA_WEBSERVER/twiki/bin/view/SIM_SECRET`
7. Click 'Subscribe to Feed' to add the feed.
8. The page should re-fresh, showing this feed as auto-located, named 'Twiki SIM_SECRET Web' and available to be deleted.
9. Click 'View Feeds' to return to your list of feeds.
10. This feed should appear in your list of SIM_SECRET feeds, and content from the feed should appear in a feed summary box.
11. Close the web browser.
12. Logout TPE 2.

A.6.5 Functional Test: Remove the last feed from your list

Test Instruction

1. Establish a session at SIM_SECRET for cudemo using TPE 1.
2. Start a web browser on Client 2.
3. Navigate to the URL `http://MYSEA_WEBSERVER/news`
4. The expected result is the MLS NEWS READER main page with only a single, subscribed feed, 'Twiki SIM_SECRET Web'.
5. Click 'Manage Feeds.'
6. Click the box to select the feed 'Twiki SIM_SECRET Web' (the only feed available).
7. Click 'Unsubscribe from Selected Feeds' to delete the feed.

8. The page should refresh, showing your list of feeds is empty.
9. Click ‘View Feeds’ to return to your list of feeds.
10. Your list of feeds should be empty; no summary boxes should appear on the page.
11. Close the web browser.
12. Logout TPE 2.

A.6.6 Functional Test: Add a feed, to existing feeds

Test Instruction

1. Establish a session at SIM_UNCLASSIFIED for mdemo1 using TPE 1.
2. Start a web browser on Client 1.
3. Navigate to the URL `http://MYSEA_WEBSERVER/news`
4. The expected result is the MLS NEWS READER main page, with pre-subscribed demo feeds showing.
5. Click ‘Manage Feeds.’
6. Input the address `http://MYSEA_WEBSERVER/twiki/bin/view/SIM_UNCLASSIFIED`
7. Click ‘Subscribe to Feed’ to add the feed.
8. The page should re-fresh, showing this feed as auto-located, named ‘Twiki SIM_UNCLASSIFIED Web’ and available to be deleted.
9. Click ‘View Feeds’ to return to your list of feeds.
10. This feed should appear in your list of SIM_UNCLASSIFIED feeds.
11. A new summary box should appear showing this feed.
12. Close the web browser.
13. Logout TPE 1.

A.6.7 Functional Test: Remove a feed

Test Instruction

1. Establish a session at SIM_UNCLASSIFIED for mdemo1 using TPE 1.
2. Start a web browser on Client 1.
3. Navigate to the URL `http://MYSEA_WEBSERVER/news`
4. The expected result is the MLS NEWS READER main page, with multiple subscribed feeds, one of which is ‘Twiki SIM_UNCLASSIFIED Web’.
5. Click ‘Manage Feeds.’
6. Click the box to select the feed ‘Twiki SIM_UNCLASSIFIED Web.’

7. Click ‘Unsubscribe from Selected Feeds’ to delete the feed.
8. The page should re-fresh, showing the feed is no longer in your list of feeds.
9. Click ‘View Feeds’ to return to your list of feeds.
10. This feed should no longer appear in your list of SIM_UNCLASSIFIED feeds.
11. Close the web browser.
12. Logout TPE 1.

A.6.8 Functional Test: Navigating and viewing subscriptions

Test Instruction

1. Establish a session at SIM_UNCLASSIFIED for mdemo1 using TPE 1.
2. Start a web browser on Client 1.
3. Navigate to the URL `http://MYSEA_WEBSERVER/news`
4. Click the ‘All Feeds’ link to view all subscribed feeds.
5. Take a moment to verify the following features of this view:
   5.a. Each news item title is accompanied by a favicon for the feed;
   5.b. Items are listed in reverse chronological order (new to old);
   5.c. Each item has a visible and correct label, appropriately colored;
   5.d. Each item contains provenance data (link to source, date published).
6. For each feed in your list of subscribed feeds (in the left-hand navigation menu), click the link.
7. Take a moment to verify the following features of this view:
   7.a. The link navigates to a page displaying items from the selected feed;
   7.b. This view shares all the features listed above, in Step 5.
8. Click the ‘Only Feeds at this Level’ link to view all subscribed feeds at your current level.
9. Take a moment to verify the following features of this view:
   9.a. Only items at your level are displayed;
   9.b. This view shares all the features listed above, in Step 5.
10. Close the web browser.
11. Logout TPE 1.

A.6.9 Functional Test: Read-down to subscriptions

Test Instruction

1. Establish a session at SIM_SECRET for mdemo1 using TPE 1.

2. Start a web browser on Client 1.
3. Verify that the left-hand navigation menu shows a list of subscribed feeds for your level, and for levels below yours.
4. Perform Test Procedure 6.8.1 Steps 3–9.
5. Close the web browser.
6. Logout TPE 1.

A.7 MLS News Reader Exception Test Details

A.7.1 Exception Test: Add a feed with blank address

1. Establish a session at SIM_SECRET for mdemo1 using TPE 1.
2. Start a web browser on Client 1.
3. Navigate to the URL `http://MYSEA_WEBSERVER/news`
4. The expected result is the MLS News Reader main page; note what feeds are subscribed in the left-hand navigation menu.
5. Click 'Manage Feeds.'
6. Click 'Subscribe to Feed' to add the feed, *i.e.*, without specifying any value in the address box.
7. The page should re-fresh, showing an error message from the MLS News Reader.
8. Click 'View Feeds' to return to your list of feeds.
9. Your feed list should be unchanged from Step 4.
10. Close the web browser.
11. Logout TPE 1.

A.7.2 Exception Test: Add a feed with blank address

1. Establish a session at SIM_SECRET for mdemo1 using TPE 1.
2. Start a web browser on Client 1.
3. Navigate to the URL `http://MYSEA_WEBSERVER/news`
4. The expected result is the MLS News Reader main page; note what feeds are subscribed in the left-hand navigation menu.
5. Click 'Manage Feeds.'
6. Add whitespace to the address box.
7. Click 'Subscribe to Feed,' to add the feed.
8. The page should refresh, showing an error message from the MLS News Reader.
9. Click 'View Feeds' to return to your list of feeds.

10. Your feed list should be unchanged from Step 4.
11. Close the web browser.
12. Logout TPE 1.

A.7.3 Exception Test: Add a feed with invalid address

1. Establish a session at SIM_UNCLASSIFIED for mdemo1 using TPE 1.
2. Start a web browser on Client 1.
3. Navigate to the URL `http://MYSEA_WEBSERVER/news`
4. The expected result is the MLS NEWS READER main page; note what feeds are subscribed in the left-hand navigation menu.
5. Click 'Manage Feeds.'
6. Add the invalid address `http://foo-nonexist` to the address box.
7. Click 'Subscribe to Feed,' to add the feed.
8. The page should refresh, showing an error related to this feed.
9. Click 'View Feeds' to return to your list of feeds.
10. Your feed list should be unchanged from Step 4.
11. Close the web browser.
12. Logout TPE 1.

A.7.4 Exception Test: Subscribe to a high feed at low (Read-up)

1. Establish a session at SIM_UNCLASSIFIED for mdemo1 using TPE 1.
2. Start a web browser on Client 1.
3. Navigate to the URL `http://MYSEA_WEBSERVER/news`
4. The expected result is the MLS NEWS READER main page; note what feeds are subscribed in the left-hand navigation menu.
5. Click 'Manage Feeds.'
6. Add, to the address box, the high address `http://MYSEA_WEBSERVER/twiki/bin/view/SIM_SECRET`
7. Click 'Subscribe to Feed,' to add the feed.
8. The page should refresh, showing an error related to this feed.
9. Click 'View Feeds' to return to your list of feeds.
10. Your feed list should be unchanged from Step 4.
11. Close the web browser.
12. Logout TPE 1.

A.7.5 Exception Test: Write-down tests

1. There are no interfaces that allow exception tests to verify write-down. No errors are provided or displayed when write-down fails, no interfaces are provided to attempt to update configuration files that are not writeable, etc.
2. This is a placeholder procedure, to note that this category of test is not absent due to any oversight.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX B: MLS News Reader Performance Tests

The following details the conditions of the performance tests used to evaluate our prototype MLS News Reader service. For all tests, we prepare the system in the following manner:

- The cache timeout is artificially increased to ensure that all feeds are fetched from some single-level cache, and no attempt is made to access the content from remote providers.
- Chronological view paging is disabled, removing the "displayed item limit" from this view. Thus, all items from every subscribed feed will be read and displayed in this view, to artificially observe performance across these dimensions.
- The caches are populated with the content for all subscribed feeds. All feeds are hosted from the MYSEA MLS Web Server, but this is largely irrelevant as content is only fetched from cache.
- The DSS client is connected, a TPE session is established at the appropriate level, and Internet Key Exchange (IKE) negotiation occurs to eliminate all one-time setup costs associated with a MYSEA session.

Tests 1–3 allow us to compare performance at a single level, as we handle different types and quantities of feeds. Tests 1, 4–5 allow us to compare performance while "reading down" to items from feeds at different levels. We use the following types of artificially generated test feeds:

- A *text feed* is a simple RSS feed (~4,425 bytes in size) containing 10 items, each with a different timestamp (*pubdate*). To exercise our chronological sorting logic non-trivially, dates are chosen at random from 1984–2011; within a feed, dates are sorted in descending order, with the first item being the most current. When processing text feeds, it follows that the image banner logic will not perform any `GET` operations.
- An *image feed* (or "good" image feed) is a text feed (~4,775 bytes in size) containing 10 items. Each item references a JPEG images (66 kb in size).
- A *mixed "bad" feed* (or "bad" image feed) is a text feed containing 10 items. Half of these feeds contain only text (~4,403 bytes in size); the other feeds (~4,753 bytes in size) each reference five JPEG images (66 kb each in size) and five GIF images (49 bytes each in size).

B.1 Index page tests

For these tests, we evaluate the index page (which provides the summary, or snapshot, view) as it processes feeds.

Test 1. At *Unclassified*, load the page subscribed to n text feeds, where $n \in \{2, 4, 8, 16, 32, 64, 128\}$.

Test 2. At *Unclassified*, load the page subscribed to n "good" feeds, where $n \in \{2, 4, 8, 16, 32, 64, 128\}$. In addition to data collected from JMeter, we quantitatively note how many images are in the random image banner.

Test 3. At *Unclassified*, load the page subscribed to n mixed "bad" feeds, where $n \in \{2, 4, 8, 16, 32, 64, 128\}$.

Test 4. At *Secret*, load the page subscribed to n *Unclassified* and n *Secret* "good" feeds, where $n \in \{1, 2, 4, 8, 16, 32, 64\}$. The page will load a total of m feeds with images, where $m \in \{2, 4, 8, 16, 32, 64, 128\}$.

B.2 Chronological view tests

For these tests, we evaluate the chronological page view, which sorts and displays all feed items. For these, we use only text feeds.

Test 5. At *Unclassified*, load the page subscribed to n text feeds, where $n \in \{2, 4, 8, 16, 32, 64, 128\}$. The page will load a total of $10n$ items.

Test 6. At *Secret*, load the page subscribed to n *Unclassified* and n *Secret* text feeds, where $n \in \{1, 2, 4, 8, 16, 32, 64\}$. The page will load a total of m feeds ($10m$ items), where $m \in \{2, 4, 8, 16, 32, 64, 128\}$.

APPENDIX C:
Feeds Used During Performance Testing

The following PHP script was used to generate our test feeds.

```
<?php
//////////////
// All we have to do here is enter our security label and the type of
// test feed the program can't (yet) predict what we're thinking!
//////////////

//////////////
// here we enter our security label

$level = "SIM_" . "UNCLASSIFIED";

// "SIM_UNCLASSIFIED" or "SIM_SECRET" or "SIM_TOP_SECRET" or
// "SIM_NATO_SECRET"
//////////////

//////////////
// Here we enter the type of text feed
// "/text_feed_" = All feeds are all text.
// "/good_feed_" = All feeds contain jpgs.
// "/bad_feed_" = 50% of feeds are all text, the other 50% are 50% jpg,
// 50% gif.

$kind = "/" . "text" . "_feed_";

// "/text_feed_" or "/good_feed_" or "/bad_feed_"
//////////////

//////////////
// Get current working directory so we create our folders
// in the right place

$thisdir = getcwd();

$level_short = substr($level,4,1);
```

```
// "U" for unclass, "S" for secret,  "T" for Top Secret,]
// "N" for Nato Secret
//////////////

$html =
"<!doctype html>
<html>
<head>
    <link rel=\"alternate\" type=\"application/rss+xml\" title=\"\" href
       =\"feed.xml\"/>
    <title>This page has a valid " . $level_short . " RSS feed</title>
</head>
<body>
    <h1>This page has a valid " .  $level_short . " RSS feed</h1>
    <a name=\"story1\">Story1</a>
    <a name=\"story2\">Story2</a>
    <a name=\"story3\">Story3</a>
    <a name=\"story4\">Story4</a>
    <a name=\"story5\">Story5</a>
    <a name=\"story6\">Story6</a>
    <a name=\"story7\">Story7</a>
    <a name=\"story8\">Story8</a>
    <a name=\"story9\">Story9</a>
    <a name=\"story10\">Story10</a>
</body>
</html>";
for($m=1; $m<129; $m++)
{
  $text_len = strlen("A valid U RSS feed ") +  /* strlen($kind) + */
      strlen($m);

  $url_len = strlen("http://mlsserver.cisrlabmlstestbed1.com/news/tests/"
      ) + strlen($level) +strlen($kind) + strlen($m) + strlen("/feed.xml"
      );

  $pref[] = "O:10:\"Pref_Store\":3:{s:9:\"sec_label\";s:" . strlen($level
      ) . ":\"" . $level . "\";s:4:\"name\";s:" . $text_len . ":\"A valid
       " . $level_short . " RSS feed " . $m . "\";s:3:\"url\";s:" .
      $url_len . ":\"http://mlsserver.cisrlabmlstestbed1.com/news/tests/"
       . $level . $kind . $m . "/feed.xml\";}\n";
  $feedpre[] = "<?xml version=\"1.0\" encoding=\"utf-8\"?>
<rss version=\"2.0\" xmlns:atom=\"http://www.w3.org/2005/Atom\">
  <channel>
    <atom:link href=\"http://mlsserver.cisrlabmlstestbed1.com/news/tests/
        $level/" . $kind . $m . "/feed.xml\" rel=\"self\" type=\"
        application/rss+xml\" />
    <title>A valid " . $level_short ." RSS feed " . $m . "</title>
    <description>A valid " . $level_short . " RSS feed " . $m . ".</
        description>
    <link>http://mlsserver.cisrlabmlstestbed1.com/news/tests/" . $level .
         $kind .  $m .   "/index.html</link>";
}
//////////////
// First make 7 folders titled 2, 4, 8, 16, 32, 64, 128
for($i=2; $i<129; $i = $i * 2)
{
    //////////////
    // If the folder is titled 2 it should have 2 xml feeds within it
    // and so on.
    for ($x = 1; $x <= $i; $x++)
    {
        if( strcmp($kind, "/good_feed_") == 0 || strcmp($kind, "/
            bad_feed_") == 0)
    {
      if($x % 2)
      {
        // Odd-numbered feeds with have images
        $pic = "will_be_overwritten";
      }
      else
      {
        // Even-numbered feeds will not have images
        $pic = "";
      }
    }
    else
    {
      $pic = "";
    }
    if(mkdir($thisdir . "/" . $i . "/" . $kind . $x , 0777, true))
    {
```

```
//////////////
// This is our default (preferences) file.
// We create 1 default file per test folder.
//////////////

$handle_default = fopen($thisdir . "/" . $i . "/default", "a");

$time = array();
for($k=0; $k<10; $k++)
{
  $time[] = mktime(rand(1,12), rand(1,60), rand(1,60), rand(1,12),
      rand(1,28), rand(1984, 2011));
}

      rsort($time);
$sort_date = array();

for($l=0; $l<10; $l++)
{
  $sort_date[] = date("D, d M Y H:i:s", $time [$l]);
}
$item = array();
for($p=0; $p<10; $p++)
{
  if( (strcmp($kind, "/bad_feed_") == 0) && ($pic != "") )
  {
    if($p % 2)
    {
      $pic = "&lt;img src=&quot;" .  $level_short . ".jpg&quot; /&
          gt;";
    }
    else
    {
      $pic = "&lt;img src=&quot;" .  $level_short . ".gif&quot; /&
          gt;";
    }
  }
  else if( (strcmp($kind, "/good_feed_") == 0))
  {
    $pic = "&lt;img src=&quot;" .  $level_short . ".jpg&quot; /&gt;
        ";
```

```
            }
                    $item[] = "
            <item>
              <title>story " . $p . "</title>
              <link>http://mlsserver.cisrlabmlstestbed1.com/news/tests/" .
                 $level . $kind . $x."/index.html</link>
              <description>Text of story "  . $p. ".
              "
                . $pic .
              "
              </description>
              <guid>http://mlsserver.cisrlabmlstestbed1.com/news/tests/" .
                 $level . $kind . $x ."/index.html#story" . $p . "</guid>
              <pubDate>" . $sort_date[$p]. " GMT</pubDate>
            </item>
          ";
          }
          $feedpo = "
    </channel>
  </rss>";
          $handle1 = fopen($thisdir . "/" . $i . "/" .  $kind  . $x . "/feed.
             xml", "a");
          $handle2 = fopen($thisdir . "/" . $i . "/"  . $kind . $x . "/index.
             html", "a");

          fwrite($handle1, $feedpre[$x-1]);
          for($t=0; $t<10; $t++)
          {
                    fwrite($handle1, $item[$t]);
          }
          fwrite($handle1, $feedpo);
          fwrite($handle2, $html);

                if( strcmp($kind, "/good_feed_") == 0 || strcmp($kind, "/
                   bad_feed_") == 0)
                {
                    $j_image = $level_short . ".jpg";
                    $g_image = $level_short . ".gif";
                    $jpg = $thisdir . "/" . $i . "/" .  $kind  . $x . "/" .
                       $j_image;
                    $gif = $thisdir . "/" . $i . "/" .  $kind  . $x . "/" .
                        $g_image;
                    copy($j_image, $jpg);
                    copy($g_image, $gif);
                }
            }
      }
      for($t=0; $t<$x; $t++)
      {
        fwrite($handle_default, $pref[$t-1]);
      }
    }
    echo "done";
    ?>
```

APPENDIX D:
Random Image Banner Application Logic

The following PHP code shows the logic used to generate our random image banner on the MLS News Reader's main page.

```php
<?php
//// begin snippet ////
function img_scraper($string)
{
    $string = html_entity_decode($string, ENT_QUOTES, 'UTF-8');
    $pattern_img = '/<img[^>]+\>/i';
    $pattern_src = '/src=[\'"]?([^\'" >]+)[\'" >]/';
    $matches = '';
    if (preg_match($pattern_img, $string , $matches))
    {
        $string = $matches[0];
        if (preg_match($pattern_src, $string, $link))
        {
            $link = $link[1];
            $link = urldecode($link);
            $test =  parse_url($link);
            if (isset($test['scheme']) === FALSE)
            {
                $link = 'http://'. HOST . '/news/' . $link;
            }
            // getimagesize doesn't seem to work with https.
            else if ($test['scheme'] == 'https')
            {
                // $link = str_replace('https', 'http', $link);
                return null;
            }
            return $link;
        }
    }
    return null;
}
//// end snippet ////

//// begin snippet ////
```

```
// We want both the SimplePie object ($sp[1]) and its
// respective security label ($sp[0]).
$image_array[] = array($sp[0], $sp[1]);
// randomize the order of the feeds
shuffle($image_array);

//// end snippet ////

//// begin snippet ////
if(!empty($image_array))
    {
        $pics = array();
        $counter = 0;
        $image_results = array();
        $image_descrip = array();
        $num_feeds = count($image_array);
        $img_feeds = array();
        $num_attempts = 0;
        $inf_loop_counter = 0;

        // If we go through all our feeds, have less than 6 pics,
        // and have done less than 10 GETs, we should loop through again
        while((count($pics) < 6))
        {
            foreach($image_array as $key => $value)
            {
                $num_attempts++;
                // break 2 gets us out of the "while" loop
                if($num_attempts === 11 || count($pics) === 6 ||
                   $inf_loop_counter === 33)
                {
                    break 2;
                }
                $value[1]->get_permalink();

                // We pick a random value of an item in the feed
                $quant =  $value[1]->get_item_quantity();

                // If the feed has no items, we skip it for obvious
                   reasons
```

```
if($quant === 0)
{
    $num_attempts--;
    continue;
}
$random_item = rand(1, $quant);
    // We check this random value
    foreach($value[1]->get_items($random_item) as $item)
                              {
        $image_url= $item->get_permalink();
        $image_title= $item->get_title();
        $image_descrip = $item->get_description();
        $scraped = img_scraper($image_descrip);

        // If our image scraper doesn't find an image,
        // we haven't yet done a costly GET, so can leave
        // the feed in $image_array. However, if we only
        // had feeds without images, we could end up in
        // an infinite loop. So we provide a counter
        // to avoid this possibility.
        if(!$scraped)
        {
            $inf_loop_counter++;
            $num_attempts--;
            if(empty($image_array))
            {
                break 3;
            }
            break;
        }
        if($value[0] === $lev)
        {
             $size = getimagesize(img_scraper(
                 $image_descrip));
        }
        else
        // If we're not at our lowest level,
        // we can only take images from the cache
        // or from MYSEA services
        {
```

```
    if(dirname(img_scraper($image_descrip)) == (
        addhttp(HOST) . '/news'))
    {
        $size = getimagesize(img_scraper(
            $image_descrip));
    }
    else
    {
        break;
    }
}
if(isset($size))
{
    list($width, $height, $type, $attr) = $size;
}
else
{
    // If we successfully scrape, but get
    // nothing, something weird is going on.
    // Since it cost us a GET and we got nothing
    // from it, we penalize the feed
    // by removing it from our image_array.

    unset($image_array[$key]);

    // If we have scraped our last feed,
    // we need to break out of our while loop and
    // call it a day.
    if(empty($image_array))
    {
        break 3;
    }
    break;
}
// 2 = JPG, 3 = PNG
$good_image = (( $type === 2) || ($type === 3));
// 1 = GIF
// Very wide images mess up the formatting
// and small pictures look bad since
// they are expanded.
```

```
                $bad_image = (($type === 1) || ((2 * $height) <
                    $width) || ($height < 20));

                if($good_image && !$bad_image)
                {
                    $pics[] = array(img_scraper($image_descrip),
                         $image_url,  $image_title, $value[0]);

                    // Remove dupes
                    // from http://stackoverflow.com/a/946300

                    $pics = array_map("unserialize", array_unique
                        (array_map("serialize", $pics)));

                    // We got a good image from the feed and
                    // move on to the next feed so as not to
                    // let one feed monopolize our images.
                    break;
                }

                // bad_images and "mystery" images end up here
                // and the entire feed gets penalized
                // i.e., is removed from the image_array.
                // might reconsider removing feeds with
                // inconveniently-sized images...
                else
                {
                    // remove feed with bad image
                    // from image_array
                    unset($image_array[$key]);

                    if(empty($image_array))
                    {
                        break 3;
                    }
                    break;
                }
            }
        }
        // If we've gone through all our feeds and still
        // have a few GETs to spare, we reshuffle the
```

```
                // image_array to keep things fresh.
                shuffle($image_array);
            }
            $num_pics = count($pics);

    //// end snippet ////
?>
```

REFERENCES

[1] J.R. Clapper. Information sharing. Center for Strategic and International Studies, January 2012. `http://csis.org/multimedia/audio-information-sharing-keynote-speaker-james-clapper`.

[2] Department of Defense. DoD Directive No. 8320.02 (Data Sharing in a Net-Centric Department of Defense), December 2004.

[3] R.J. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. John Wiley & Sons, New York, 2001.

[4] D.E. Denning. A Lattice Model of Secure Information Flow. *Communications of the ACM*, 19 (5):236–243, 1976.

[5] D.E. Bell and L. LaPadula. Secure Computer System: Unified Exposition and Multics Interpretation. Technical Report MTR-2997, The MITRE Corporation, March 1976.

[6] T.E. Levin, C.E. Irvine, C. Weissman, and T.D. Nguyen. Analysis of Three Multilevel Security Architectures. *Proc. of the Computer Security Architecture Workshop*, pp. 37–46, November 2007.

[7] J. Alves-Foss, W. Harrison, P. Oman, and C. Taylor. The MILS Architecture for High-Assurance Embedded Systems. *International Journal of Embedded Systems*, 2(3/4): 239–247, 2006.

[8] M.H. Kang and I. Moskowitz. Design and Assurance Strategy for the NRL Pump. *IEEE Computer*, 31(4):56–64, April 1998.

[9] O.S. Saydjari. Multilevel Security: Reprise. *IEEE Security & Privacy*, 2(5):64–67, 2004.

[10] T. Hinke. The trusted approach to multilevel security. In *Proc. of the Computer Security Applications Conference*, pp. 335–341, December 1990.

[11] C.E. Irvine, R.R. Schell, and M.F. Thompson. Using TNI concepts for near-term use of high assurance database management systems. In *Proc. of the Fourth RADC Database Security Workshop*, pp. 335–341, April 1991.

[12] C. Owen, D. Grove, T. Newby, A. Murray, C. North, and M. Pope. PRISM: Program Replication and Integration for Seamless MILS. In *IEEE Symposium on Security and Privacy*, pp. 281–296. IEEE Computer Society, Los Alamitos, CA, USA, 2011.

[13] T. Holzmann, Y. Fu, S. Hirshfield, R. Cunningham, and J. Randall. Commercial cross-domain XML guard. Technical Report AFRL-IF-RS-TR-2005-380, Air Force Research Lab Rome, November 2005.

[14] A. Thummel and K. Eckstein. Design and implementation of a file transfer and web services guard employing cryptographically secured XML security labels. In *Proc. of the IEEE Information Assurance Workshop*, pp. 26–33, June 2006.

[15] L. Sauer, M. Maschino, J. Morrow, and M. Mayhew. Towards achieving cross domain information sharing in a SOA-enabled environment using MILS and MLS technologies. In *Proc. of the IEEE Military Communications Conference (MILCOM 2009)*, pp. 1–5, October 2009.

[16] E. Bersack. Implementation of a HTTP (web) server on a high assurance multilevel secure platform. Master's thesis, Naval Postgraduate School, December 2000.

[17] K. L. Ong, T. D. Nguyen, and C. E. Irvine. Implementation of a Multilevel Wiki for Cross-Domain Collaboration. In *Proc. of the 3rd International Conference on Information Warfare and Security (ICIW 2008)*, pp. 293–304, 2008.

[18] Yahoo! Media RSS Module. `http://video.search.yahoo.com/mrss`. Last accessed 3/3/12.

[19] L. Sikos. *Web Standards: Mastering HTML5, CSS3, and XML.* Apress, Inc., 2011.

[20] Blogbridge. `http://www.blogbridge.com`. Last accessed 3/2/11.

[21] Rssowl. `http://www.rssowl.org`. Last accessed 3/2/11.

[22] Amphetadesk. `http://www.disobey.com/amphetades`. Last accessed 3/2/11.

[23] AgileRSS. `http://www.agilerss.com`. Last accessed 3/2/11.

[24] rss2mail. `http://www.nongnu.org/rss2mail`. Last accessed 3/2/11.

[25] Newspipe. `http://newspipe.sourceforge.net`. Last accessed 3/2/11.

[26] feed2imap. `http://home.gna.org/feed2imap`. Last accessed 3/2/11.

[27] PopUrls® Genuine Aggregator. `http://popurls.com`. Last accessed 3/3/11.

[28] Planet Mozilla. `http://planet.mozilla.org`. Last accessed 3/3/11.

[29] Feedweaver. `http://feedweaver.net`. Last accessed 3/2/11.

[30] xfruits. `http://www.xfruits.com`. Last accessed 3/2/11.

[31] Blogsieve. `http://www.blogsieve.com`. Last accessed 2/28/12.

[32] Feedrinse. `http://www.feedrinse.com`. Last accessed 2/28/12.

[33] Yahoo! Pipes. `http://pipes.yahoo.com/pipes`. Last accessed 2/28/12.

[34] Bloglines. `http://www.bloglines.com`. Last accessed 3/2/11.

[35] Google reader. `http://reader.google.com`. Last accessed 3/2/11.

[36] Feedshow. `http://reader.feedshow.com`. Last accessed 3/2/11.

[37] feed on feeds. `http://feedonfeeds.com`. Last accessed 3/2/11.

[38] Tiny Tiny RSS. `http://tt-rss.org`. Last accessed 3/2/11.

[39] zfeeder. `http://zvonnews.sourceforge.net`. Last accessed 3/2/11.

[40] lylina rss aggregator. `http://lylina.sourceforge.net`. Last accessed 3/2/11.

[41] Rnews Feed Aggregator. `http://rnews.sourceforge.net`. Last accessed 3/2/11.

[42] Urchin. `http://urchin.sourceforge.net`. Last accessed 3/2/11.

[43] Lilina News Aggregator. `http://getlilina.org`. Last accessed 3/2/11.

[44] Moonmoon. `http://moonmoon.org`. Last accessed 3/2/11.

[45] My News Crawler. `http://www.mynewscrawler.com`. Last accessed 3/2/11.

[46] PHP RSS Aggregator Script. `http://www.phprssreader.com`. Last accessed 3/2/11.

[47] Planet Feed Reader. `http://planetplanet.org`. Last accessed 3/2/11.

[48] rawdog. `http://offog.org/code/rawdog.html`. Last accessed 3/2/11.

[49] curn. `http://software.clapper.org/curn`. Last accessed 3/2/11.

[50] gpodder. `http://gpodder.org`. Last accessed 3/2/11.

[51] Zhuangzi, S. Hamill, and J.P. Seaton. *The Essential Chuang Tzu.* Shambhala Publications, 1998.

[52] C.E. Irvine, T.D. Nguyen, D.J. Shifflett, T.E. Levin, J. Khosalim, C. Prince, P.C. Clark, and M.A. Gondree. MYSEA: the Monterey Security Architecture. In *Proc. of the 2009 ACM workshop on Scalable trusted computing (STC'09)*, pp. 39–48. ACM, New York, NY, USA, 2009.

[53] T.D. Nguyen, M.A. Gondree, D.J. Shifflett, J. Khosalim, T.E. Levin, and C.E. Irvine. A Cloud-Oriented Cross-Domain Security Architecture. In *Proc. of the IEEE Military Communications Conference (MILCOM 2010)*, pp. 441–447, 2010.

[54] E. Bertino, S. Castano, E. Ferrari, and M. Mesiti. Specifying and enforcing access control policies for XML document sources. *World Wide Web*, 3(3):139–151, June 2000.

[55] A. Gabillon and E. Bruno. Regulating access to XML documents. In *Proc. of the 15th annual working conference on Database and application security (DAS '01)*, pp. 299–314. Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[56] M. Kudo and S. Hada. XML document security based on provisional authorization. In *Proc. of the 7th ACM conference on Computer and Communications Security (CCS'00)*, pp. 87–96. ACM, New York, NY, USA, 2000.

[57] E. Damiani, S. De Capitani di Vimercati, S. Paraboschi, and P. Samarati. A fine-grained access control system for XML documents. *ACM Transactions on Information and System Security*, 5:169–202, May 2002.

[58] W. Fan, C.Y. Chan, and M. Garofalakis. Secure XML querying with security views. In *Proc. of the 2004 ACM SIGMOD international conference on Management of data (SIGMOD '04)*, pp. 587–598. ACM, New York, NY, USA, 2004.

[59] J.S. Park and G. Devarajan. Fine-grained and scalable approaches for message integrity. In *Proc. of the 40th Annual Hawaii International Conference on System Sciences (HICSS '07)*, p. 280c. IEEE Computer Society, Washington, DC, USA, 2007.

[60] A.G. Stoica and C. Farkas. Secure XML views. In *Proc. of the 16th International Conference on Data and Applications Security (IFIP'02)*, pp. 133–146, July 2002.

[61] G. Kuper, F. Massacci, and N. Rassadko. Generalized XML security views. In *Proc. of the tenth ACM symposium on Access control models and technologies (SACMAT '05)*, pp. 77–84. ACM, New York, NY, USA, 2005.

[62] S. Oudkerk, I. Bryant, A. Eggen, and R. Haakseth. A proposal for an XML confidentiality label and related binding of metadata to data objects. In *RTO-MP-IST-091: Information Assurance and Cyber Defence*. NATO Research and Technology Organisation, November 2010.

[63] K. Zeilenga. XEP-0258: Security Labels in XMPP. `http://xmpp.org/extensions/xep-0258.html`. Last accessed 2/28/12.

[64] P. Millard, P. Saint-Andre, and R. Maijer. XEP-0060: Publish-Subscribe. `http://xmpp.org/extensions/xep-0060.html`. Last accessed 2/28/12.

[65] T.H. Hinke and M. Schaefer. Secure Data Management System. Technical Report RADC-75-266, Systems Development Corporation, November 1975.

[66] National Computer Security Center. *A Guide to understanding object reuse in trusted systems*. 1992.

[67] A. Wilhelm. Twitter bans repetitive tweets to block spam, October 2009. `http://thenextweb.com/2009/10/14/twitter-bans-repeat-tweeting-block-spam/`. Last accessed 3/1/12.

[68] L. Bouzereau. *Star Wars: The Annotated Screenplays*. Star Wars Series. Ballantine Books, 1997.

[69] National Academy of Sciences. *The 1986 Workshop on Integrated Database Development for the Building Industry*. National Academy Press, Woods Hole, MA, June 1986.

[70] SimplePie. `http://simplepie.org`. Last accessed 3/1/12.

[71] D. Shafik, L. Mitchell, and M. Turland. *PHP Master: Write Cutting Edge Code*. O'Reilly & Associates Inc, 2011.

[72] Simplepie documentation: set_stupidly_fast(). `http://simplepie.org/wiki/reference/simplepie/set_stupidly_fast`. Last accessed 3/14/2012.

[73] The White House. Executive Order No. 13526 (Classified National Security Information), 75 Fed. Reg. 707, January 5, 2010.

[74] Department of Defense. *National Industrial Security Program Operating Manual (NISPOM)*, February 28, 2006. DoD 5220.22-M.

[75] Information Security Oversight Office. *Marking classified national security information*, December 2010.
`http://www.archives.gov/isoo/training/marking-booklet.pdf`.

[76] pheedo. `http://www.pheedo.com`. Last accessed 3/4/2012.

[77] L. Rainey, C. Poggi, and L. Wittman. *Futurism: An Anthology*. Henry McBride Series in Modernism and Modernity. Yale University Press, 2009.

[78] Apache JMeter. `http://jmeter.apache.org`. Last accessed 3/17/12.

[79] C.V. Clausewitz, C.F.N. Maude, and J.J. Graham. *On War*. Wilder Publications, 2008.

[80] Instapaper. `http://www.instapaper.com`. Last accessed 3/14/2012.

[81] Instapaper: Information for Publishers. `http://www.instapaper.com/publishers`. Last accessed 3/14/2012.

[82] B. Savage. *The PHP Playbook*. Marco Tabini & Associates, Inc., 2011.

[83] A.P.J.A. Kalam and A. Tiwari. *Wings of Fire: An Autobiography*. Universities Press, 1999.

THIS PAGE INTENTIONALLY LEFT BLANK

Initial Distribution List

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Steve LaFountain
   National Security Agency
   Fort Meade, MD

4. Victor Piotrowski, National Science Foundation
   Arlington, VA

5. Dr. Mark Orwat
   National Reconnaissance Office
   Chantilly, VA

6. Dr. Salim Zafar
   National Reconnaissance Office
   Chantilly, VA

7. Dr. Mark Gondree
   Naval Postgraduate School
   Monterey, CA

8. Dr. Cynthia E. Irvine
   Naval Postgraduate School
   Monterey, CA

9. Thuy D. Nguyen
   Naval Postgraduate School
   Monterey, CA